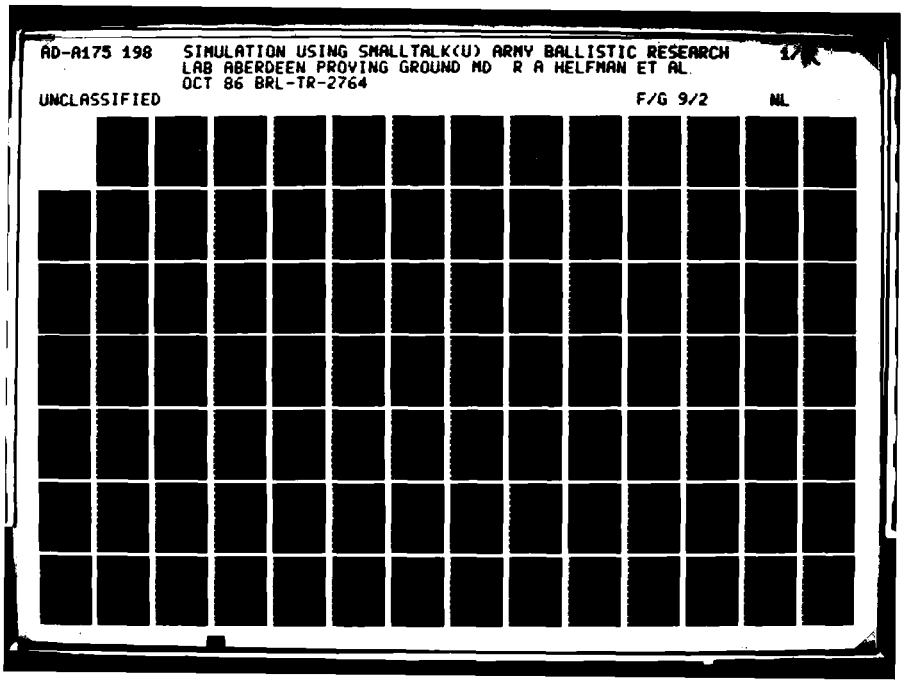
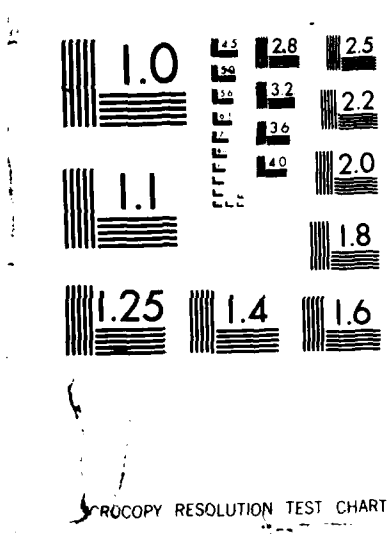

AD-A175 198 SIMULATION USING SMALLTALK(U) ARMY BALLISTIC RESEARCH
LAB ABERDEEN PROVING GROUND MD R A HELFMAN ET AL.
OCT 86 BRL-TR-2764
UNCLASSIFIED
F/G 9/2
NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
ROCOPY RESOLUTION TEST CHART

(12)

US ARMY MATERIEL COMMAND

AD

TECHNICAL REPORT BRL-TR-2764

AD-A175 198

SIMULATION USING SMALLTALK

Richard A. Helfman
Mark H. Ralston
J. Robert Suckling

October 1986

US ARMY BALLISTIC RESEARCH LABORATORY
ABERDEEN PROVING GROUND, MARYLAND

Destroy this report when it is no longer needed.
Do not return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U. S. Department of Commerce, Springfield, Virginia 22161.

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute indorsement of any commercial product.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TECHNICAL REPORT BRL-TR-2764	2. GOVT ACCESSION NO. AD-A175198	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SIMULATION USING SMALLTALK		5. TYPE OF REPORT & PERIOD COVERED FINAL 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard A. Helfman Mark H. Ralston J. Robert Suckling		8. CONTRACT OR GRANT NUMBER(s) ACN 82562
9. PERFORMING ORGANIZATION NAME AND ADDRESS U.S. Army Ballistic Research Laboratory ATTN: SLCBR-VL Aberdeen Proving Ground, MD 21005-5066		10. PROGRAM ELEMENT, PROJECT, TASK, AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Ballistic Research Laboratory ATTN: SLCBR-DD-T Aberdeen Proving Ground, MD 21005-5066		12. REPORT DATE October 1986 13. NUMBER OF PAGES 170
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

simulation
object oriented programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Object oriented languages have been used successfully in such areas as simulation, systems programming, graphics, and Artificial Intelligence (AI). Object oriented programming has become increasingly popular in the 1980's. SMALLTALK is an object oriented language developed by Xerox, that has features particularly suited to simulation.

(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The purpose of this paper is twofold, the first is to acquaint the reader with t' concept of object oriented programming; the second is to describe how the obj.ct oriented language SMALLTALK was used for a simulation application at the U.S. Army Ballistic Research Laboratory (BRL).

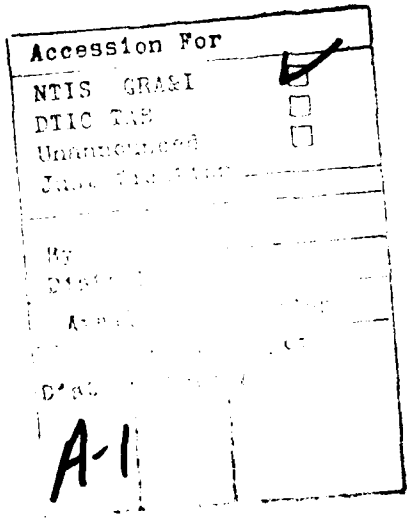
The common theme in object oriented languages is objects. Objects possess properties of procedures (functic·s, subroutines) and data since they perform computations and store information. This dual role contrasts with procedural languages such as C, FORTRAN, and PASCAL which separate procedures from data. Objects communicate by sending messages to other objects. Similar objects can be grouped to form an entity called a class. A class represents a generic kind of object, and can be thought of as a pattern or template for that kind of object. The classes themselves can be objects, and classes can be grouped to form a hierarchy of classes. An object in one class can inherit the behavior of objects in its superclasses. Thus one can easily define classes that are "nearly alike", since inheritance eliminates the need for duplicating redundant information.

The U.S. Army Quartermaster School, Ft. Lee, VA, commissioned the BRL to perform a study of the Graves Registration (GRREG) Service. The thrust of the study was to evaluate the GRREG requirements of the future battlefield and evaluate the ability of the GRREG system to meet these requirements. The study provided 1) a base line analysis of the ability of the present system to handle conventional and contaminated remains, and 2) an analysis of several alternatives, including changes in force structure, equipment, and GRREG procedures. The recommendations of the study are intended to provide the Logistics Community a direction for changes in graves registration doctrine, procedures, and organizations.

The GRREG services are best described as a network of queues. The network is rather complicated: consisting of several hundred individual queues that are interconnected either in series or in parallel. The network will be described in three levels of detail, with the basic level consisting of the individual task queues, the intermediate level consisting of the three types of collecting points (initial, intermediate, cemetery), and the top level showing the flow from one collecting point to another.

Selected sections of the code will be examined in detail to illustrate the various stages in the life of a GRREG worker, including creating a worker, assigning a task, performing a task, and returning to idle status. Relevant parts of the code will be reproduced as needed with the discussion, and the complete code is included in the appendix.

CONTENTS



LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF EXHIBITS

I. Introduction

Simulation is a powerful and widely used analytic tool. It is often the only useful tool for problems that defy mathematical formulation. There are many situations which cannot be solved mathematically due to either the stochastic nature, or to the complexity, or to the interactions of the elements of the model, and simulation can often be used to obtain relevant answers.

An essential part of any simulation is a representation of the system under study. This representation leads to the construction of a computer program that "describes" the model to be studied, and there are several commercial simulation packages on the market, including SIMSCRIPT, GPSS, SLAM, SIMULA, and GASP that have been widely used.

Object oriented programming has become increasingly popular in the 1980's. Object oriented languages have been used successfully in such areas as simulation, systems programming, graphics, and Artificial Intelligence. SMALLTALK is an object oriented language, developed by Xerox, that has features particularly suited to simulation.

The purpose of this paper is twofold: the first is to acquaint the reader with the concept of object oriented programming; the second is to describe how the language SMALLTALK was used for a simulation application for the US Army.

The paper is organized in several parts. Chapter II discusses the history and basic features of object oriented programming. Chapters III- V describe the syntax of the SMALLTALK language. Chapters VI-VIII describe how SMALLTALK was used in a simulation for the US Army Quartermaster School, and chapters IX and X contain conclusions and a summary.

II. Object Oriented Programming

This chapter will discuss some of the generic features of object oriented programming, starting with a brief history, and then explaining some of the key features including objects, message passing, and inheritance. Much of the latter discussion is based on the article "Object Oriented Programming: Themes and Variations".[1]

1. History

The roots of object oriented programming can be traced to SIMULA[2], a simulation language developed by Sperry Rand Corporation. Although other systems have shown some object oriented tendencies, the explicit awareness of the idea, (including the term "object oriented"), came from the Smalltalk effort at Xerox.[3] The language SMALLTALK[4] is the first major interactive, graphic based implementation of object oriented programming.[1] Many key concepts of object oriented programming can be seen in a variety of other languages today. Frame based AI languages such as KEE[5], FRL[6] and UNITS[7] use inheritance of properties and/or values. Objects and message passing occur in several LISP dialects: T[8], XLISP, COMMONLOOPS.[9]

---

1. Stefek, Mark and Bobrow, Daniel. "Object-Oriented Programming: Themes and Variations," AI Magazine, Winter 1986, pp. 40-62.

2. Dahl, O.J. and Nygaard, K. "SIMULA - An Algol Based Simulation Language," Comm. ACM, No. 9, 1966, pp. 671-678.

3. Rentsch, Tim. "Object Oriented Programming," Dept. of Computer Science, Working Paper, UCLA, n.d.

4. Goldberg, Adele and Robson, D. "Smalltalk-80: The Language and its Implementation," Addison-Wesley, 1983.

5. Fikes, R. and Kehler, T. "The Role of Frame-Based Representation in Reasoning," Comm. ACM, Vol. 28 No. 9, 1985, pp. 905-920.

6. Goldstein, I.P. and Roberts, R.B. "NUDGE, A Knowledge-Based Scheduling Program," IJCAI-1977, pp. 257-263.

7. Stefik, Mark. "An Examination of a Frame-Structured Representation System," IJCAI-1979, pp. 845-852.

8. Rees, J.A., Adams, N.I., and Meehan, J.R. "The T Manual," Yale Univ. Technical Report, Jan. 1984.

9. Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L.,Stefik, M. and Zdybel, F. "CommonLoops: Merging Common Loops and Object- Oriented Programming," ISL-85-8, Xerox PARC, Aug, 1985.

2. Objects and Messages

The common theme in object oriented languages is objects. Objects possess properties of procedures (functions, subroutines) and data since they perform computations and store information. This dual role contrasts with procedural languages such as C, FORTRAN, and PASCAL which separate procedures from data.

Objects communicate by sending messages to other objects. When an object receives a message, it typically performs some action. The action might include numerical computations, storing or updating local information, or sending further messages. Message passing can result in a kind of indirect procedural call. Instead of calling a procedure to compute some value, one sends an object a message to perform some computation.

The actions an object takes when it receives a message are called its method for that message. A method roughly corresponds to a procedure in ordinary programming languages.

3. Classes

Similar objects can be grouped to form an entity called a class. A class represents a generic kind of object, and can be thought of as a pattern or template for that kind of object. The real numbers might be grouped into a class, say class Reals. Rather than specifying the behavior of each real number, one only needs to define how an arbitrary member of class Reals will respond to various messages. To carry this further, the classes themselves can be objects, and classes can be grouped to form a hierarchy of classes. For example the class of Numbers might contain Reals and Imaginaries, and Reals could have subclasses Integers and Floats.

4. Inheritance

With a hierarchy of classes, an object in one class can inherit the behavior of objects in its superclasses. This has several important consequences. First, it greatly simplifies the task of specifying how an object will respond to a message. Continuing with the Numbers example, if the class Reals has a method (procedure) for the message 'isPositive' (i.e. "is this a positive number"), then 'isPositive' need not be repeated in class Integers nor in class Floats, since each will inherit all methods in its superclasses. Thus one can easily define classes that are "nearly alike", since inheritance eliminates the need for duplicating redundant information.

5. Data Abstraction

The idea behind data abstraction is that of defining a pattern or template for objects.[10] Objects can then be declared to be of a particular pattern and can inherit all the attributes and behavior defined by the pattern. As in Simula, such a pattern is called a class. (cf. the term package is used in Ada). Data abstraction allows individual objects to inherit the properties of the classes to which they belong.

Data abstraction localizes (and conceals) the details of an object. Conceptually, each class of objects resides on its own machine or computer, and objects communicate with each other only by passing messages. In effect, the objects partition the system's memory into disjoint blocks. Since all objects in a class have the same properties, the code for a class can be examined once to identify those properties. If a change is necessary, it need be made only once in the class definition rather than once for each object in that structure. Thus data abstraction localizes (and conceals) the details of generating and manipulating objects.

The purpose of data abstraction is to permit the use of objects without any knowledge of the details of implementation. An example of data abstraction in our simulation is the class Queue. The user sees such operations as 'addTo' or 'removeFrom' or 'length' and need only know their visible behavior.[11] Hidden from the user are the details of how a queue is stored internally, or how the messages are implemented.

III. Introduction to Smalltalk

This and the next two chapters describe some of the syntactical features of SMALLTALK that are useful in a simulation environment, and many useful examples are provided to give a more concrete understanding of the language.

Smalltalk is an object oriented language. The basic entities of the language are objects. Objects have a private memory and are capable of sending and receiving messages. When an object

---

10. Cohen, A. Toni. "Data Abstraction, Data Encapsulation, and Object Oriented Programming," Dept. of Computer and Information Sciences, Working Paper, Univ. of Delaware, n.d.

11. Shankar, K.S. "Data Structures, Types, and Abstractions," Computer, April, 1980, pp. 67-77.

receives a message, it typically performs some sequence of operations. For example, in Smalltalk, numbers are objects which can respond to several kinds of messages, including 'abs', 'exp', 'gamma', and 'ln'. The following table gives a few of these messages and their meaning.

Message	Meaning
abs	absolute value
exp	e raised to the power
gamma	gamma value
ln	natural logarithm
reciprocal	the arithmetic reciprocal
sign	-1 or 0 or 1 depending on whether the object is negative, zero, or positive
sqrt	square root if the object is positive

1. Syntax and Examples

The syntax of a Smalltalk expression is:

```
object   message   <optional arguments>
```

where object is the receiver of the message, and the message may contain optional arguments.

For example, the number 3 is an object and it can respond to various messages. Thus to compute the square root of 3, one would send the message 'sqrt' to the object '3'.

Expression	Result
3 squared	9
3 sqrt	1.7320
3 gamma	2
3 reciprocal	0.3333
3 sign	1

Some messages such as '+', '-', '*', and '/' have arguments. Thus to add 3 and 4 together, one would send the object '3' the message '+' with the argument '4'.

Expression	Result
3 + 4	7
3 - 2	1
3 * 5	15
3 / 6	.5

Numbers can also receive messages concerning magnitude, e.g. '<', '>', '=', and 'max:' .

Expression	Result
3 < 2	false
3 = (6 / 2)	true
3 max: (4 max: 5)	5

On the first line, the object '3' is sent the message '<' with the argument '2'. The result is the object 'false'. In the second line, the object '3' is sent the message '=' with the argument '(6 / 2)'. The result is the object 'true'. The last line is evaluated from right to left. The object '4' is sent the message 'max:' with argument '5', and the result is the object '5'. Then the object '3' is sent the message 'max:' with the object '5'. The result is the object '5'.

Note that Smalltalk contains two boolean objects, 'true' and 'false'. These two objects can receive messages such as: '&' and '|' , which represent "and" and "or" respectively.

Expression	Result
(3 < 4) & (1 > 5)	false
(3 < 4) | (1 > 5)	true

On the first line, the object '3 < 4' is sent the message '&' with argument '1 > 5'. Now '1 > 5' will return 'false', and '3 < 4' will return 'true', so that when 'true' is sent the message '&' with argument 'false', the overall result will be 'false'.

2. Objects

An object is a basic entity in Smalltalk. In fact, everything is an object in Smalltalk. To completely describe an object one must specify:

1. its private memory,

2. the messages it can receive, and

3. what it does when it receives a legal message.

Each object is capable of carrying out a certain set of operations. The nature of the operation depends upon the type of object. Objects representing numbers compute arithmetic functions. Objects representing data structures store and retrieve information.

In Smalltalk each integer is an object and can respond to messages such as '+' and '-' and '*' and '/' to name a few. Other examples of objects include:

numbers
character strings
sets
queues
dictionaries
arrays
rectangles
files
i/o streams

3. Classes

To simplify the description of objects, Smalltalk allows similar objects to be grouped together to form a class. All objects in the class have the same type of private memory and respond to the same messages in a similar way. Consider the integers:

```
... -3 -2 -1 0 1 2 3 ...
```

These objects are grouped together to form the class Integer. All objects in the class Integer respond in a similar way to the same set of messages, such as '+' '-' '*' or '/' .

A class can be thought of as a template or blueprint describing all objects in the class. The template must specify:

1. what private memory the object has

2. the set of messages the object can receive

3. what operations the object performs when it receives a legal message.

Note however that defining a class does not create any objects in that class. It merely specifies what a certain group of objects look like if they are ever created. To actually create an object, one must send the message 'new' to the appropriate class.* For example, suppose one wants to create a vector of size 5. One would send the message 'new: 5' to Array.

```
Array new: 5
```

Since in practice one usually wants to refer to this array by some convenient name, say v, one would probably use the following assignment statement instead of the above.

```
v <-   Array new: 5
```

This has now created an object v which is a member of class Array. Some examples of its behavior are shown below.

```
Example              Result
-------              ------

v size               5
v print              #( nil nil nil nil nil )
v at: 1 put: 3       3
v print              #(  3  nil nil nil nil )
```

When v gets the message 'size', it returns the object '5', and then when it gets the message 'print', it returns the string shown on the second line. The # sign indicates an array, and its contents are given in parentheses. Since no one has yet put anything into v, its elements are all undefined objects, textually represented by 'nil'. On line 3, the object '3' is placed into position '1', and then v is printed again.

---

* There are a few exceptions to this, e.g. Numbers and Symbols have already been created when Smalltalk is started up.

4. Subclasses and Superclasses

Smalltalk comes with a rich hierarchy of predefined classes, some of which will be described in the next section. The user can define his own additional hierarchy of classes to fit specific applications, and some examples will be presented later.

One class can be a subclass of another, in which case the subclass inherits all the memory and messages from the superclass. Consider two classes, A and B, where B is a subclass of A, as indicated below.

```
Class A
        Class B
```

One also says that A is a superclass of B.

Suppose one declares an object, say b, to be a member of Class B:

```
b <- B new
```

Now suppose one sends a message to b:

```
b 'any_message'
```

If the description of Class B contains a clause for 'any_message', then b will undertake the appropriate action as defined in its class. If on the other hand, 'any_message' is not a legal message in Class B, the description of Class A will be searched for 'any_message'. If it is there, then b will execute the operations as defined in Class A. If, however, it is not there, then the search for 'any_message' continues up through all superclasses of A. (If the search fails, then an error exists, and the undefined object 'nil' is returned). Note that this search strategy implies that messages can be redefined in subclasses. Any message defined in a subclass will override any definition given in a superclass.

5. Pseudo-variables

A pseudo-variable is a name that refers to a Smalltalk object, but unlike a regular variable it can not appear on the left hand side of an assignment statement. Some pseudo-variables in the system are constant and always refer to the same object:

Pseudo-variable	Meaning
nil	refers to a special undefined object
true	refers to an object representing logical truth
false	refers to an object representing logical falsity

There are two other pseudo-variables 'self' and 'super' whose values change depending on where they occur. These two will be described later.

6. Messages

Objects send messages to other objects or to themselves. The only way an object can interact with another is through messages. The general syntax for messages is:

```
object message   <optional arguments>
```

The object is called the receiver of the message. The message may include optional arguments. The syntax can be divided into three cases.

1. Messages without arguments
2. Messages with keyword arguments
3. Arithmetic messages

a. **Messages without arguments.** The simplest form for a message is:

```
object message
```

Examples include:

```
4 sqrt
4 squared
4 abs
4 print
```

b. **Messages with keyword arguments.** The general form for messages with keyword arguments is:

```
object keyword1: argument1  keyword2: argument2  ...
```

The receiver of the message is 'object'. Keywords are identified by a trailing colon. A message may contain several

```
keyword: argument
```

pairs. Examples of messages with keywords are:

```
v <- Array new: 5
v at: 3 put: 0.444
```

c. **Arithmetic messages.** The third type* of message occurs mainly in arithmetic expressions. Examples are

```
3 + 4
sum - 1
index <-  bound
```

7. Assignments

A constant will always refer to the same object, but a variable name may refer to different objects at various times. An assignment expression has the form

```
variable <- expression
```

The object referred to by the variable is changed when the expression is evaluated. For example,

```
limit <- 19
reportTitle <- 'Smalltalk Report'
sum <- 3 + 4
```

8. Returned Values

A message provides for two-way communication. The message is sent to the receiver along with any arguments, and the receiver performs certain operations. In addition, the receiver also returns an object to the sender of the message. If the

---

* This syntax is preferable to the form given above, e.g. 3 +: 4.

message occurred in an assignment statement, then the returned object will be the new value of the variable in that assignment statement. Thus the expression

```
sum <- 3 + 4
```

makes 7 the new value of the variable named sum. Even if no information needs to be communicated back to the sender, a receiver always returns an object, and this tells the sender that the response to the message is complete.

9. Blocks and Arguments

A block consists of a sequence of expressions surrounded by square brackets, and blocks are used in many of the control structures in the language. A block can be thought of as a deferred set of actions to be performed at some later time. Syntactically, a block is an object and can send and receive messages. When a block expression is encountered, the statements in the brackets are not executed immediately; rather they are remembered. The value returned is an object that can later execute the expressions when sent a message to do so. The execution of the block will take place when the block receives certain messages, such as 'value'. For example,

```
amount <- amount + 1
```

and

```
[amount <- amount + 1]  value
```

and

```
b <- [amount <- amount + 1]
b value
```

all have identical effects.

One example of simple control structure is repetition or looping, accomplished by sending the message 'timesRepeat:' to an Integer. The Integer will respond by sending the message 'value' to the block as many times as it own value indicates. Thus,

```
4 timesRepeat: [amount <- amount + 1]
```

is equivalent to

```
temp <- [amount <- amount + 1]
temp value
temp value
temp value
temp value
```

Blocks may have one or more arguments specified by identifiers preceded by colons at the beginning of the block. The general form for a block with one argument is:

```
[ :argument | one or more expressions ]
```

Consider the following example.

```
sum <- 0
b <- [ :x | sum <- sum + x ]
```

To add 5 to sum, one sends the message 'value: 5' to b:

```
b value: 5
```

A block may have more than one argument, as shown in the example below.

```
[ :key :value | v at: key put: value ]
```

The arguments first appear preceded by colons, and then after the vertical bar, they are used without colons.

10. Conventions

The names of objects begin with small letters, while the names of classes begin with capital letters. Also the names of messages usually begin with small letters. This convention is enforced by the language. Note also there is a convention to run words together, capitalizing all but the first to enhance readability. For example,

```
myNewObject
```

rather than

```
my_new_object
```

This convention is of course not enforced by Smalltalk.

IV. Predefined Classes in Smalltalk

This chapter covers some of the predefined classes in SMALLTALK, and the casual reader may wish to skim this material. The following sections refer to the chart given below in Exhibit 1. This chart lists many of the predefined classes in SMALLTALK. The subclass structure is indicated by indenting to the right. For the readers convenience, portions of the chart under discussion will be reproduced close by the written dialog, and an arrow will highlight which class is under discussion.

Exhibit 1. Smalltalk Classes

```
Object
        UndefinedObject
        Symbol
        Boolean
                True
                False
        Magnitude
                Char
                Number
                        Integer
                        Float
                Radian
                Point
        Random
        Collection
                Bag
                Set
                KeyedCollection
                        Dictionary
                                Smalltalk
                        SequenceableCollection
                                Interval
                                LinkedList
                                File
                                ArrayedCollection
                                        Array
                                        String
        Block
        Class
        Process
```

1. Object

Since everything is an object in SMALLTALK, at the top of the list there is the class of all objects whose name is Object. The class Object has several subclasses. The first three are called UndefinedObject, Symbol, and Boolean. By convention, the names of classes begin with capital letters, and the objects within the class begin with small letters.

a. **Undefined Object.** This class has only one member, denoted 'nil', and it is used to represent undefined values. By default, SMALLTALK initializes all objects to 'nil'. Also, 'nil' is the object returned in an error situation. For example, the expression

```
true  sqrt
```

would return 'nil'.

b. **Symbol.** This is the class used to represent the print names of objects in the system. Its members are created automatically by SMALLTALK.

c. **Boolean.** This class has two subclasses: True and False. Class True has only one member, 'true', and class False has only one member, 'false'.

------- Exhibit 1. (Partial) -------

```
Object
        UndefinedObject
        Symbol
        Boolean
                True
                False
=>      Magnitude
                Char
                Number
                        Integer
                        Float
                Radian
                Point
```

---

d. **Magnitude.** The next class to be discussed is Magnitude. This is the class of all objects possessing a linear ordering. All messages in this class are defined in terms of the basics, '<', '=', and '>'.

Examples.	Result.
3 < 5	true

Class Magnitude has several subclasses as seen above, and Smalltalk automatically creates all members of the subclasses shown.

(1) **Char.** Class Char contains the objects representing single ASCII characters. They are written by preceding the character desired with a dollar sign, for example: $a $B $4 $$.

(2) **Number.** Class Number contains the two subclasses: Integer and Float which represent integer and floating point numbers respectively.

(3) **Radian.** Class Radian is used to represent radians. Only radians will respond to messages such as 'sin' and 'cos'. Numbers can be converted to radians by passing them the message 'radians'. Similarly radians can be converted to numbers by sending them the message 'asFloat'. Radians are normalized to be between 0 and 2*pi.

Examples	Result
0.5236 radians sin	0.5
0.5 arcSin asFloat	0.5236

(4) **Point.** Class Point contains pairs of numbers representing coordinates. They are represented by placing the @ sign between two numbers.

Examples	Result
(0@0) dist: (3@4)	5.0
(1@2) + (3@4)	(4@6)

------- Exhibit 1. (Partial) -------

```
Object
        UndefinedObject
        Symbol
        Boolean
        Magnitude
=>      Random
        Collection
```

---

e. **Random.** The class Random provides protocol for random number generation. Sending the message 'next' to a member of Random results in a Float between 0.0 and 1.0 randomly distributed.

```
Example                 Result
-------                 ------

ran <- Random new
ran next                0.683
ran next                0.466
ran next: 3             #(0.095  0.166  0.745)
```

The first line creates 'ran' as a member of class Random. Since no seed is specified, the default one will be used. The message 'new randomize' sent to Random will create an object with a random seed. Sending the message 'next: 3' generates three random numbers.

------- Exhibit 1. (Partial) -------

```
Object
        Magnitude
        Random
=>      Collection
                Bag
                Set
                KeyedCollection
                        Dictionary
                                Smalltalk
                        SequenceableCollection
```

---

f. **Collection.** This class represents groups of objects, such as Sets or Arrays. The different types of forms in class Collection are distinguished by several characteristics including:

1. whether the size of the collection is fixed or unbounded,

2. whether the collection is ordered,

3. the methods for retrieving and inserting objects into the collection.

For example, an Array is a collection with a fixed size and an ordering indexed by integer keys.

(1) **Bag/Set.** Bags and Sets are unbounded unordered collections, and their elements are not indexed by any keys. The difference between a Bag and a Set is that an element can occur repeatedly in a Bag but not in a Set. For example, suppose b is an object in class Bag containing four elements:

```
Example          result
-------          ------

b print          Bag( ball bat glove bat   )
b asSet print    Set( ball bat glove  )
```

------- Exhibit 1. (Partial) -------

```
Object
        Collection
                Bag
                Set
=>              KeyedCollection
                        Dictionary
                                Smalltalk
                        SequenceableCollection
```

---

(2) **KeyedCollection.** Elements in this collection are pairs of the form:

```
key   value
```

In the case of class Array, the key is called the index and is

usually an Integer. The message

```
at: key
```

will return the item in the collection having the given key. The message

```
at: key put: value
```

is used to insert an item, and

```
removeKey: key
```

is used to delete an item.

(a) **Dictionary.** Class Dictionary is a subclass of KeyedCollection. Both the key and value portions of an element can be any object, although commonly the keys are instances of Class Number or Symbol. In the example below, a Dictionary of opposites called 'opp' is created.

```
Example                    Result
-------                    ------

opp <- Dictionary new
opp at: #hot put: #cold
opp at: #stop put: #go
opp at: #big  put: #little

opp size                   3
opp print                  Dictionary
                           ( #hot  @ #cold
                             #stop @ #go
                             #big  @ #little )

opp at: #big               #little
```

(1) Smalltalk. The class Smalltalk contains one member 'smalltalk'. This object serves several functions. First, it provides global communication between all objects. Second, it is used to modify various parameters used by the Smalltalk system. Third, it can pass commands to the Unix shell.

------- Exhibit 1. (Partial) -------

```
Object
        Collection
                Bag
                Set
                KeyedCollection
                        Dictionary
                                Smalltalk
                =>      SequenceableCollection
                                Interval
                                LinkedList
                                File
                                ArrayedCollection
                                        Array
                                        String
```

(b) **SequenceableCollection.** This class contains objects in KeyedCollection that are indexed by integer keys. Since there is a definite fixed order for elements in this class, it is possible to refer to the first and last elements of an object. Elements in this class also respond to the messages 'sort' which will return the object sorted from smallest to largest, and 'reversed' which will return the object with the elements in reverse order.

(1) Interval. The members of this class represent sequences of numbers in an arithmetic sequence, either ascending or descending.

Expression	Meaning
(1 to: 5)	#(1 2 3 4 5)
(1 to: 5 by: 2)	#(1 3 5)
(5 to: 1 by: -2)	#(5 3 1)
(.3 to: .7 by: .1)	#(0.3 0.4 0.5 0.6 0.7)

Used with the message 'do:', a control structure similar to "do" or "for" loops can be obtained. For example:

```
(1 to: 10) do: [:x | x print]
```

will print the integers 1 through 10.

(2) LinkedList. Objects in this class represent stacks or queues. The objects have a fixed order, but no definite size. Elements can only be added or removed from the

ends, i.e. either the beginning or the end. An example of a queue is a line of people in a bank. People enter the bank and join the end of the line, and when their turn comes for service they leave the beginning of the line. An example of a stack is a pile of letters on a desk. When a new letter comes in, it goes on the top of the pile, and letters are (usually) removed from the top.

Message	Meaning
addFirst: object	the object is added to the beginning of the collection
addLast: object	the object is added to the end of the collection
removeFirst	remove the first element
removeLast	remove the last element

(3) File. The elements of class File are stored on an external medium, typically a disk. Objects in this class respond to messages such as:

```
'open: filename'
'read'
'write: object'
```

------- Exhibit 1. (Partial) -------

```
Object
        Collection
                KeyedCollection
                        SequenceableCollection
                                Interval
                                LinkedList
                                File
                        =>      ArrayedCollection
                                        Array
                                        String
```

---

(4) ArrayedCollection. The class ArrayedCollection contains two subclasses: Array, and String. The difference between them is that while the values in class Array can be any objects, in class String they must be from class Char. Textually, arrays are represented by a pound sign preceding the array, and strings are represented by placing single quotes around the entire string.

```
Example                    Result
-------                    ------

a <- #(10 12 14)
a size                     3
a at: 3                    14
b <- 'string'
b size                     6
b at: 2 put: $p            'spring'
```

```
------- Exhibit 1. (Partial) -------

Object
        Collection
=>      Block
        Class
        Process

-------------------------------------
```

g. **Block.** Blocks are used in many control structures in the language. A block represents a deferred sequence of operations. Textually they are represented by square brackets surrounding a sequence of Smalltalk expressions. Blocks are objects in the system and can respond to messages. When a block is encountered, the statements in the brackets are not executed immediately, rather an object is created. The sequence of operations that a block describes will be performed when the object receives the message 'value'. For example,

```
Example                                    Result
-------                                    -------

increment <- [ index <- index + 1 ]
index <- 0
increment value                            1
increment value                            2
```

Blocks can be passed arguments with the message 'value: object'. For example,

```
[:x | x + 3]  value:  6
```

will result in the value 6 being passed in for x. The result of the block will hence be 9. The expression ":x" appearing in the block says that "x" is the parameter in the block.

h. **Class.** Users can define their own classes by sending messages to Class. The message consists of the name of the new class followed by its definition. For example, a new class called Probability has been defined for use in our simulation studies. The actual form of the class will be discussed in the next chapter, but the definition starts with

```
Class Probability
        [ <definition of this class> ]
```

i. **Process.** Processes are created by the system or by sending the message 'newProcess' or 'fork' to a block. They can not be created directly by the user.

V. Classes and Messages

The previous chapter dealt with the hierarchy of predefined classes in SMALLTALK. This chapter starts by examining how new classes can be created, (specifically the class Probability that was introduced above), and then covers two pseudo-variables, self and super, and ends with an example illustrating how SMALLTALK handles returned values.

1. The Message 'new'

Objects are the basic components of the Smalltalk system. Messages allow interactions between the components of the system. Every object in Smalltalk is a member of a class. The members of a class all have the same message interface: the class describes how to carry out the operations available through that interface.

Objects are created by sending messages to classes. Most classes respond to the message 'new' by creating a new member of themselves. For example,

```
Array new: 5
```

returns an object that is a member of class Array having 5 elements. The object created can respond to the same messages as any other member of class Array.

2. Class Descriptions

The description of a class has five parts.

1. the name of the class
2. the class hierarchy

3. the private memory of each element

4. the set of legal messages understood by the class, and

5. the operations performed when a legal message is received.

Suppose one wants to define a class called Probability, which will be used to generate random numbers from various probability distributions such as Normal and Uniform. The various probability distributions will be subclasses of Probability as seen below.

```
Object
        Probability
                Uniform
                Normal
                Binomial
                Exponential
```

3. Example: Class Probability

The definition of Probability starts with:

```
Class Probability     :Object
```

Our new class will generate random numbers and pass them to an appropriate subclass. One local variable, say randnum, will be needed to hold the random number generated. The list of local variables is placed between vertical bars.

```
Class Probability     :Object
        | randnum |
```

Notice that spacing, tabs, and carriage returns may be used freely to improve readability.

The rest of the class definition is a block. The block contains pairs of the form:

```
message expressions
```

These pairs are separated by vertical bars.

Members of the class Probability will respond to two messages: 'initialize' and 'next'. Members of class Probability must first receive the message 'initialize', which will create an object for random number generation. The message 'next' will pass a random number to an appropriate subclass for processing. Consider the complete definition of class Probability. (Recall

that since Probability is a subclass of Object, it inherits all the messages defined in Object, so they need not be listed again unless the user wishes to redefine them.

```
Class Probability     :Object
        | randnum |
[ initialize
        randnum <- Random new.
| next
        ^self sample: randnum next.
]
```

As mentioned earlier, any member of Probability must first receive the message 'initialize' which creates a new instance of the random number generator (with default seed). That random object will be called randnum. The message 'next', which will pass a random number to a subclass, will be fully described later. Note that the symbol '^' means return the object created by this expression, and that 'self' refers to the object that received the message (in this case the object that received the message 'next'). The exact behavior of this message will become evident when the subclass Uniform is presented.

4. Example: Class Uniform

Now that one has defined Probability, one can add subclasses for various probability distributions. The easiest one is Uniform. This class will generate uniform random numbers on an interval [a,b]. Since class Random generates uniform random numbers on the interval [0,1], class Uniform need only ask Probability for such a number, say x, and then perform the translation:

```
a + ( x * ( b - a ) )
```

A member of class Uniform requires two local variables, a and b, to hold the endpoints, and it will respond to two messages: 'from: start to: stop' and 'sample: x' The first will store the endpoints a and b, and the second will ask Probability for a random number and them perform the above translation. The definition of Uniform follows.

```
Class Uniform  :Probability
        | a b |
[ from: start to: stop
        (start < stop)
                ifTrue:
                        [ a <- start.
                          b <- stop.]
                ifFalse:
                        [ self error:
                                "illegal interval"
                        ]

| sample: x
        ^ a + ( x + ( b - a ) ).
]
```

5. Messages to Uniform and Probability

At this point, two new classes have been defined. The class definitions are templates specifying how members of the class will behave. To understand the behavior of a member of Uniform (and of Probability), an example will be traced through in detail. Suppose a study involves a task whose duration is Uniform on the interval [5 9].

First one creates a member of class Uniform:

```
u <- Uniform new
```

Next, one initializes:

```
u <- initialize
```

Since class Uniform does not respond to the message 'initialize', but its superclass Probability does, the message 'initialize' is sent to Probability. A random number generator 'randnum' will be created.

To generate uniform random numbers on the interval [5 9], perform the following:

```
u <- from: 5 to: 9
```

The message 'from: start to: stop' would respond with start = 5 and stop = 9. The test '(start < stop)' is either true or false, and one of the two blocks is executed. Since 5 is less than 9, Uniform will store 5 and 9 into a and b respectively.

The message

```
u next
```

will now generate a uniform random number in the interval [5 9]. Again, Uniform does not respond to the message 'next', but Probability does. So the expression

```
^ self sample: randnum next
```

is evaluated. The evaluation starts at the right with:

```
randnum next
```

Now randnum is a member of class Random, so sending it the message 'next' will result in a random Float in the interval [0 1]. This random object then becomes the argument for the message 'sample:'. Thus the message:

```
sample: randnum next
```

is sent to 'self'. The pseudo-variable self refers to the original receiver of the original message, which in this case is 'u'. (Recall that the original message was ' u next '). In effect then, the message becomes:

```
u sample: < a random object>
```

Now u will respond to this message by evaluating

```
^ a + ( x + ( b - a ) )
```

The random object generated above will be substituted for x, and a and b have been assigned values 5 and 9, so the overall effect is to create a random object uniform on the interval [5 9].

Now that the object has been created, where does it go? The '^' means to return the object generated by the expression, so that the message

```
sample: x
```

in Uniform will return that uniform number. This object is then the value of the expression

```
self sample: randnum next
```

in Probability. The '^' on that line returns the object from the message next. (Note that if the two '^' had not been used, the random object would have still been created, but not passed back). Thus

```
u next
```

will generate the desired uniform random number.

Other examples of user defined classes will be presented later.

6. Pseudo-variables 'self' and 'super'

Messages can be sent to the pseudo-variables self and super. When a message is sent to self, it will go to the original receiver of the original message. Consider the following example in which two classes, One and Two, are defined. Class Two is a subclass of One.

```
Class One :Object
[ test
        1 print.
| pass
        self test.
]

Class Two :One
[ test
        2 print.
]
```

One creates members of each class.

```
member1 <- One new
member2 <- Two new
```

Then messages are sent to the new objects.

Expression	Result
member1 test	1
member2 test	2
member1 pass	1
member2 pass	2

Sending the message 'test' to either member1 or member2 will print a '1' or a '2' as one would expect. Sending the message 'pass' to member1 will cause the expression

```
self test
```

to be evaluated. The variable self refers to member1, so the message 'test' is sent to member1 and a '1' is printed. However, when 'pass' is sent to member2, since members of Class Two do not respond to that message, it goes instead to its superclass, which is class One. Again, the expression

```
self test
```

is evaluated, but this time the variable self refers to the original receiver, member2. Thus a '2' is printed.

The variable super refers to the superclass of the class containing the line in which super was used.

7. Returned Values

When an object receives a message, its class description will tell it what operations to perform. When it finishes processing the message, it will always return some object. The default value is the name of the receiver. If some other object is needed, then one or more return expressions '^' should be included in the class description. For example,

```
Class A
[compute
        3+4
]

Class B
[compute
        ^3+4
]
```

One creates members of both classes,

```
a <- A new
b <- B new
```

and send the message 'compute' to each.

Example	Result
a compute	a
b compute	7

When object 'a' gets the message 'compute', it calculates '7', but by default returns its name.

VI. Simulation Example

1. Introduction

In 1984, the US Army Quartermaster School commissioned the Ballistics Research Laboratory (BRL) to conduct a Graves Registration (GRREG) study. The thrust of this study was to evaluate the GRREG requirements of the future battlefield and analyze the ability of the GRREG system to meet these requirements. The study provided 1) a base line analysis of the ability of the present system to handle conventional and contaminated remains, and 2) an analysis of several alternatives, including changes in force structure, equipment, and GRREG procedures. The recommendations of the study are intended to provide the Logistics Community a direction for changes in graves registration doctrine, procedures, and organizations. A large computer simulation was written in Smalltalk in order to perform the analysis.

2. Background of Graves Registration

a. **Description of Service.** The Graves Registration Program provides for essential search, recovery, collection, and disposition of the remains of deceased US, allied and enemy personnel in an area of conflict where the prompt return of remains to the continental United States is not possible. Disposition of remains, according to current doctrine, is by burial in temporary military cemeteries. The Graves Registration Program is a logistics function under the auspices of the Quartermaster Corps. In a theater of operation, graves registration collection points are established in the Brigade Support Area. Additional collection points are established in the Division and Corps rear areas. The temporary military cemetery is established in the COMMZ or Corps rear. Current doctrine requires that units transport the remains of deceased soldiers to the nearest collection point. From there, graves registration personnel tentatively identify the remains and evacuate them, through intermediate collection points to the temporary cemetery. At the cemetery, operated by a Graves Registration Company, personnel remove personal effects from the remains for shipment to next of kin, and bury the remains.

The US Army Quartermaster Corps has responsibility for the graves registration program. This responsibility includes the organization of units to carry out graves registration functions, acquisition and training of MOS 57F (Graves Registration Specialist) personnel, the development of requirements for new items of equipment to support graves registration operations, and the development of graves registration doctrine.

The graves registration program involves four major functional areas. They are search and recovery, identification, burial and personal effects processing. All of these functions are carried out in the theater of operations. Personal effects are shipped to next of kin at the earliest possible time.

Graves registration personnel may carry out search and recovery missions in cases where a unit is unable to recover their dead, where a unit has been forced to bury remains in a hasty/temporary grave site, where an aircraft has been downed, to police the battlefield of enemy dead, or in any situation where other units are unable to recover the remains of U.S. servicemen from an area of operations. Search and recovery missions are time consuming and labor intensive. These missions sometimes force elements of a graves registration unit to operate over large geographical areas.

The identification function is carried out by graves registration personnel at a recovery site, a graves registration collection point or at a temporary cemetery. Every effort is made to completely identify remains as soon after death and as close to the place of death as possible. Experience has shown that timely identification is a significant factor in reducing the number of unknowns in a conflict. All tasks associated with documenting identifications and reporting this information will be considered as part of the identification function in this analysis.

All remains processed as part of a graves registration program are buried in the theater of operations in temporary military cemeteries. Burial is either in individually marked graves or a common grave if mass burial procedures are in effect. Under the graves registration program all cemeteries and grave sites in the theater of operations are considered temporary. The program calls for the eventual return of all remains to next of kin or military cemeteries in the United States unless a permanent military cemetery is authorized by specific legislation. Remains in hasty/temporary graves in the theater of operations are consolidated in temporary military cemeteries if possible.

Current graves registration doctrine and procedures are general in nature and oriented toward the conventional environments of past conflicts. Little attempt has been made in recent years to capitalize on current technology for identifying, reporting and processing remains.

During peacetime, the graves registration system is not used. Peacetime manpower and fiscal constraints have forced the Army to place graves registration units in the Reserve Component and graves registration elements have been removed from many active unit tables of organization and equipment (TOE). Peacetime deaths of servicemen are handled by the current death program, which emphasizes civilian mortuary services and contract support. Because of this, very few graves registration personnel are in the active force, graves registration procedures have not been kept current and problems posed by future battlefield environments have not been addressed.

b. **Organizations and Equipment.** Graves registration assets are organized into units ranging in size from the GRREG Battalion to GRREG Team Augmentations. At the lowest echelon, graves registration support is provided by teams, sections and platoons attached to supply and service companies or field service companies. These GRREG elements are organized into collection points that provide for search, recovery, initial identification and evacuation of remains. These collection points are not organized or equipped to perform burial. All graves registration support to divisions is provided by augmentation to the divisional Supply and Service Company. Divisional GRREG capability is strictly a wartime augmentation.

The Graves Registration Company carries out the final identification of remains and operates the temporary military cemetery where remains are buried. The Cemetery Company is also organized to perform search and recovery missions and to operate a collection point. Personal effects are also processed by the Cemetery Company prior to being sent to the personal effects depot for temporary storage and shipment to legal recipients.

c. **Doctrine.** Current doctrine for graves registration is illustrated by the flow of remains shown in Figure 1. Units have the responsibility of evacuating remains to the appropriate graves registration collection point. This evacuation is normally accomplished by using organic unit transportation assets. However, any available transportation may be utilized. Evacuation from teams, squads, platoons and companies may be routed through the appropriate battalion headquarters. Remains are unloaded at each echelon in an effort to keep organic unit transportation assets within a units' area of operation. Much of the current GRREG organization and doctrine is dictated by transportation requirements.

Any transportation assets may be used to evacuate remains with the exception of ration trucks. From the losing unit, transportation will normally be organic company or battalion vehicles and aircraft. Once the remains are in graves registration channels, evacuation of remains becomes the responsibility of the graves registration unit. All graves registration units have authorized organic vehicles which may be used for evacuation of remains; however, it must be kept in mind that these vehicles are also required to carry out search and rescue missions, and perform unit administrative tasks. Evacuation of remains within graves registration channels, therefore, depends upon requests for nonorganic transportation and the availability of back haul transportation assets. Doctrine provides guidelines for the transportation of remains within the theater. Remains must be covered at all times while being transported. Remains must be escorted while being evacuated to insure that personal effects are safeguarded and that the remains receive proper treatment while enroute. The vehicle transporting remains must be covered at all times and remains inside the vehicle should not touch each

other. This precludes stacking remains one on top of each other in a vehicle and limits the number of remains that can be transported in one vehicle. Utilizing litters the maximum number of remains that can be transported in a 2 and 1/2 ton cargo truck under the constraints of this doctrine is 24.

Doctrine states that identification should be carried out as soon as possible after death and as close to the scene of death as possible. Remains recovered by GRREG personnel on a search and recovery mission are identified at the recovery site if possible. Early identification is felt to be the key to eliminating unknowns. Various identification media are used and doctrine prescribes what combinations are acceptable for positive identification. It must be remembered, however, that identification media which are used as sole source evidence of identification may be wrong. For this reason, current doctrine stresses the use of multiple identification sources to confirm the identity of remains.

Burial is the only accepted disposition method for remains under current graves registration doctrine. Remains are buried in individually marked graves at consolidated temporary military cemeteries in the theater of operations. The intent of current GRREG doctrine is to discourage the use of small scattered cemeteries and consolidate the burial of remains as much as possible. Doctrine prohibits the use of isolated/hasty graves unless their use is absolutely unavoidable e.g. where a unit is unable to evacuate their dead and are being forced to move. Consolidated cemeteries are required by doctrine for many reasons. Consolidation makes it easier to carry out the return of remains program and the organization and basis of allocation of graves registration companies makes it impossible to have decentralized burial within a theater of operations. Consolidation also makes the care and maintenance of cemeteries easier, limits the possibilities that a burial site could be lost and makes it less likely that a cemetery would fall into enemy hands particularly since current doctrine places temporary military cemeteries in the COMMZ.

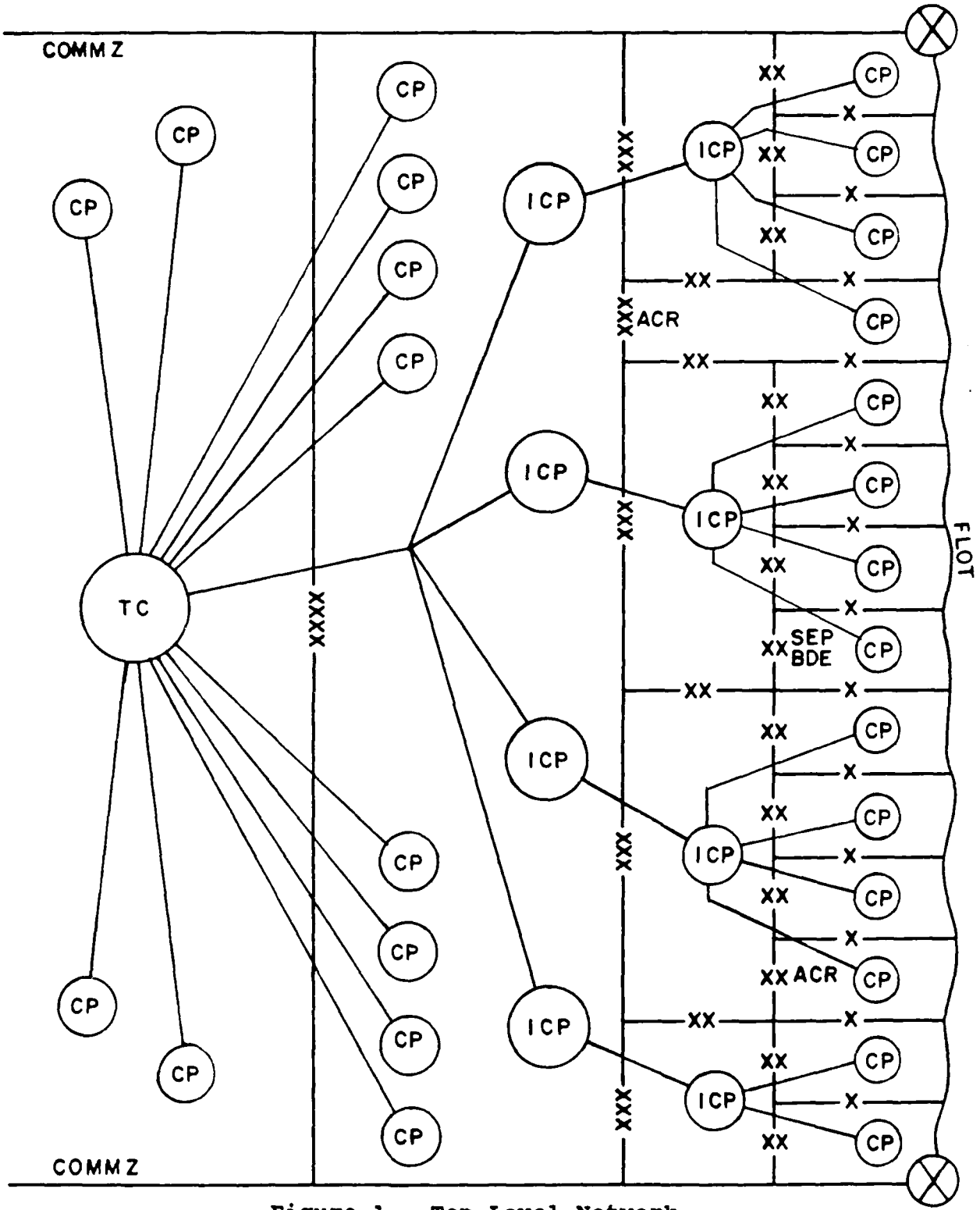


Figure 1. Top Level Network

VII. GRREG Queuing Network

1. Introduction

The graves registration organizations in a Corps are best described as a network of queues where remains await processing. These queues form networks, where the output of one becomes the input of another. The network is rather complicated: consisting of several hundred individual queues that are interconnected either in series or in parallel. The network will be described in three levels of detail, with the basic level consisting of the individual queues, the intermediate level consisting of the three types of collecting points (initial, intermediate, cemetery), and the top level showing the flow from one collecting point to another. Figure 1 illustrates the queues and networks in the corps slice of the theater at this top level. Except for the remains of personnel who die in the COMMZ and are brought directly to the cemetery for processing, all remains in the theater will pass through a minimum of two collecting points prior to burial.

a. Definitions. The GRREG queuing network forms a directed connected graph of arcs and nodes, (see Figure 2), with tokens passed along the arcs through the nodes. The tokens represent bodies or trucks, and each node represents a task to be performed on tokens and a queue where the tokens wait their turn for processing. The meaning of these terms depends on the level of detail in the network. At the top level, the nodes (circles) represent the collecting points, the arcs (lines) represent the connecting roads, and the tokens represent the trucks carrying bodies. At the intermediate level, the nodes represent individual tasks from the basic task list, the arcs represent movement from one task to the next, and the tokens represent the individual bodies at the collecting point.

Tokens are created by a generator (source) node. Each generator node has one arc leading to a task node's queue. Here the tokens wait their turn for processing. Examples of process (task) nodes are unloading trucks and taking finger prints. After the processing is completed the token travels on an arc to the next queue. This pattern is repeated until a final (sink) node is reached. An example of a sink node is a temporary cemetery plot. The sink node's queues hold tokens that represent the throughput of the GRREG services.

b. Network Structure. As mentioned before, the network can be viewed at three different levels. The description of the GRREG network will start at the top level with some basic definitions; then move to the intermediate level and a detailed discussion of the three types of collecting points; and conclude at the basic level with an examination of the various queue parameters.

2. Top Level Network

The top level nodes are the collection points:

1. Initial Collecting Point [CP]
2. Intermediate Collecting Point [IP]
3. Temporary Cemetery [TC]

and the low level network defines these nodes in more detail. A simple high level network example is shown below.

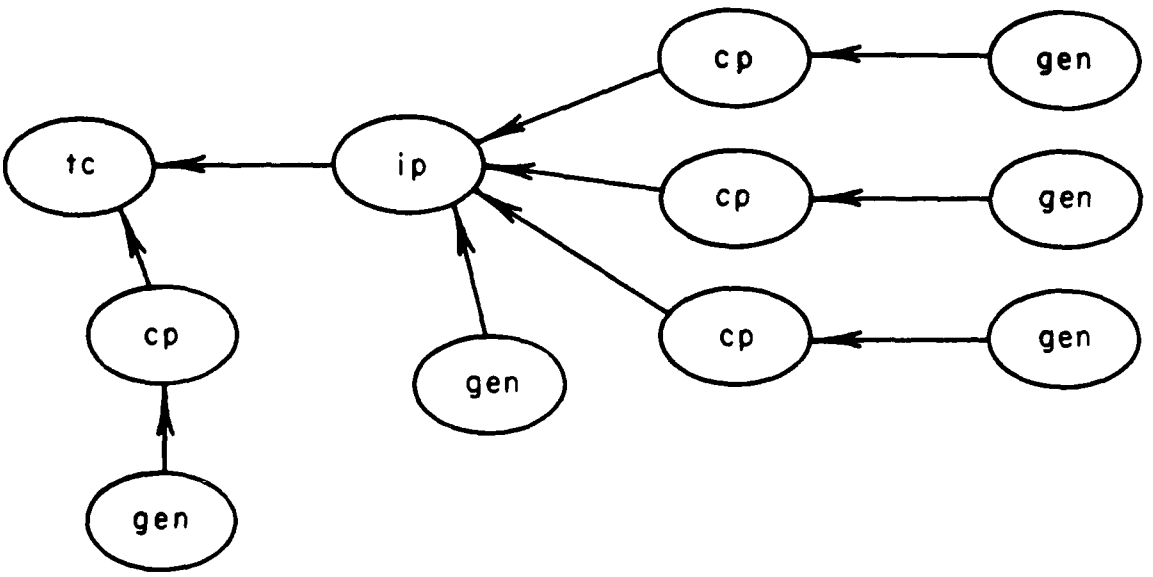


Figure 2. Example High Level Network

Each generator ([gen]) creates a work load of tokens, which consists of trucks filled with a random number of bodies. These tokens pass through collection point nodes on trucks until they

reach the Temporary Cemetery node ([tc]).

3. Intermediate Level Networks

The intermediate level networks represent the three types of collecting points (initial, intermediate, temporary cemetery). See Figures 3, 4, and 5. Here each token represents a truck full of bodies. These trucks start in the field. After the drive from the pick up point the trucks line up at the in-truck-queue. Here a pool of workers is assigned the task of unloading the truck. This pool contains several workers, with one or more workers on the truck, and the remainder (in pairs) on the ground to carry the bodies to the processing location. Here the bodies receive an evacuation number from one of the workers. This body is not physically moved, but is added to the the identification and personal effects queue.

Each collection point node has a limited capability to do processing, which is a function of the number of workers assigned to the collection point. Each task node requires one or more workers to perform the given task. When multiple workers are assigned to a task node, task characteristics determine whether the work is performed in parallel or in series. For example, tasks like loading (unloading) trucks need workers in pairs for each body to be loaded (unloaded) at one time, plus one worker in the truck. Tasks like identification require only one worker per body, while other tasks can only be done by at most one worker at any given time. An example of the latter is filling out the convoy list.

Most of the arcs in the collection points and the temporary cemeteries are simple and represent serial task queues. The exceptions are the branching, joining, and forking of arcs at nodes, to be explained below.

Branching occurs when a token can be put on one of several queues after service. This happens, for example, after the body has received an evacuation number. If the body has already been processed through an initial cclecting point, then the next task is to check the records to be sure there are no errors in processing up to this point. However, if the body has not been processed then the complete identification process must be carried out.

Forking occurs when a token is split and put on two or more queues. An example of this is can be seen at the temporary cemetery, where the holes in the ground are prepared while the body goes through final processing.

Joining occurs when a node waits for all parts of a forked token to arrive before processing continues. After the above holes and the final processing are completed, the body is ready for placing in the hole.

The nodes at the end of the arcs represent tasks to be performed by workers. These nodes consist of a queue of incoming tokens (these tokens are either trucks or bodies). This is where the tokens wait for a turn to be processed. These nodes also require one or more workers to do the processing to the bodies. Some task nodes can have more than one worker at a time (e.g. n workers can perform the ID task on n bodies), while other tasks are restricted to one worker (e.g. the Evac task), thus only one body at a time. The hardest tasks for worker allocation are the loading and unloading of the trucks, as described above, which consume two workers per body and one or more extra workers on the truck.

The actual processing of bodies is done at the level of tasks. This requires that needed resources be allocated to the task for specified time period, and then released back to the system for other tasks to consume.

To illustrate in more detail, the following are needed by the simulation to process a task.

1. a body
2. one or more workers to be consumed while the body is being processed.
3. some storage for the body and the worker (this holds the resources until the task is completed).
4. the limits on the number of workers required (as above).
5. a delay time for the execution of the task (this is the amount of time needed to complete the task).
6. arcs for the disposition of the body for its next task.

Note that 1) and 2) are consumable resources for the simulation, 3) can be forgotten until there is no more computer resources, and 4), 5), and 6) are constraints that differentiate the tasks.

The way these work is as follows. For each collection point there is a fixed set of workers. These workers are allocated to each task that meets the above needs for processing. When a task is ready to run, the body and the worker(s) are stored in a task object for storage in the time queue. After the delay time is consumed by the simulation, the task is run to free its resources. These are:

1. the body, which is placed on the next task's input queue.

2. the worker(s), which can get another body for this task or start a new kind of task.

The task list for an initial collecting point is given in Table 1. The source code for those tasks is given in Exhibit 2. Each task contains 6 methods (startUpTask, max, min, taskTime, prefix:, and next). The startUpTask method creates an instance and returns the name of the individual task. The methods max and min return the maximum and minimum number of workers for that task, while taskTime returns the average time required for the task. Note that prefix: echelon will return a string containing the echelon level and the name of the task that is used for printed reports. The message next informs the scheduler what the worker is doing, and informs the body that it has left the task queue and is now being processed.

4. Parameters for Basic Level Queues

The behavior of an individual queue is controlled by the choices made for a small set of parameters. These parameters will be examined as they apply to the various queues in the network.

a. **Arrival Parameters.** The calling population (casualty workload) for the GRREG model is finite; limited by the intensity and nature of the battle and the troop population. The simulation was run well past the last battle (i.e. no arrivals) to determine the time needed to work off the backlog.

Some queues experienced only bulk arrivals, (occurring whenever trucks arrived with bodies). Other queues had no bulk arrivals, and some had both bulk and single arrivals.

The arrival rate for bulk arrivals changed daily and depended upon battle conditions and troop populations in the vicinity. For some queues, the arrival rate was the sum of the departure (throughput) rates of one or more previous queues in the chain.

b. **Service Parameters.** Each queue in the network represents one of the tasks from the basic task list for the collection point. (See tables 1, 2, and 3). The service times for each task are independent and normally distributed. The number of servers (MOS 57F workers) at each service center changes throughout the simulation. A 'worker to task' scheduler assigns workers to individual tasks based on several factors including task priority and queue backlog. The worker stays only until task completion, at which time he is reassigned to either the same task or possible another task. Thus tasks may get no workers assigned, or may get one or more workers.

c. **Queue Discipline.** Queue discipline is first come, first served, and queue capacity is assumed to be infinite. However, for some excursions, balking was allowed at the truck arrival queues whenever the backlog reached a critical peak. The trucks would then proceed to the next higher echelon collecting point and try to join the input queue.

TABLE 1. Task List Initial Collecting Point

Task Time per Remain (min)	Task
2	Unload remains
5	Assign an evac number and record
55	Check ID tags, field medical card, prepare statement of recognition, record of recovery (if necessary), inventory PE and fingerprint
10	Place remains, documents and PE in human remains pouch and move to holding area
5	Prepare convoy list
5	Miscellaneous record keeping
2	Load on transportation

TABLE 2. Task List Intermediate Collecting Point

Task Time per Remain (min)	Task
2	Unload remains
5	Assign evac number and record
25	Compare remains with documentation and fingerprint
5	Move remains to holding area
5	Record on convoy list
2	Load remains on transportation

TABLE 3. Task List Temporary Cemetery

Task Time per Remain (min)	Task
2	Unload remains
5	Check evac number and PE seal
5	Move to processing area
5	Assign processing number and record
15	Compare remains and PE with documentation
20	Remove clothing and examine
15	Fingerprint remains
30	Perform detailed ID; consists of anatomical, dental, and/or skeletal charting, photography and comparison of evidence with records; assumed that this 30 minutes is the time for all types of ID cases averaged over every remain processed
5	Shroud remains
10	Prepare plates, tags and attach
5	Move remains to holding area
10	Dig grave site (mechanical digging)
10	Move remains to grave site
20	Prepare internment and plot records and 3x5 card
30	Place remains in grave and cover (manual)
15	Prepare and ship PE

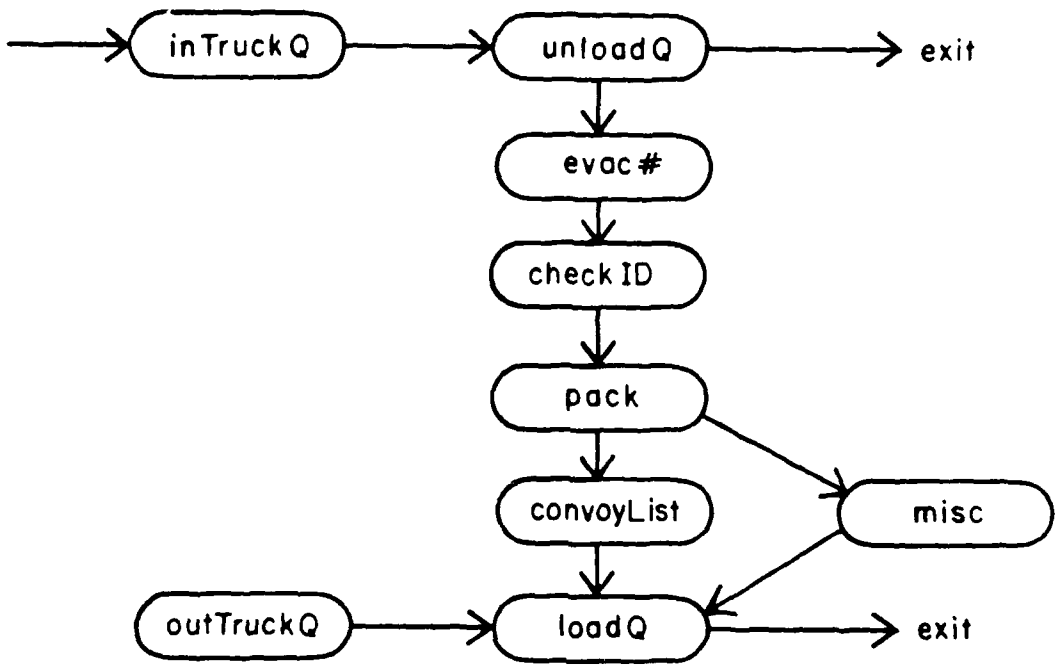


Figure 3. Initial Collecting Point Task Network

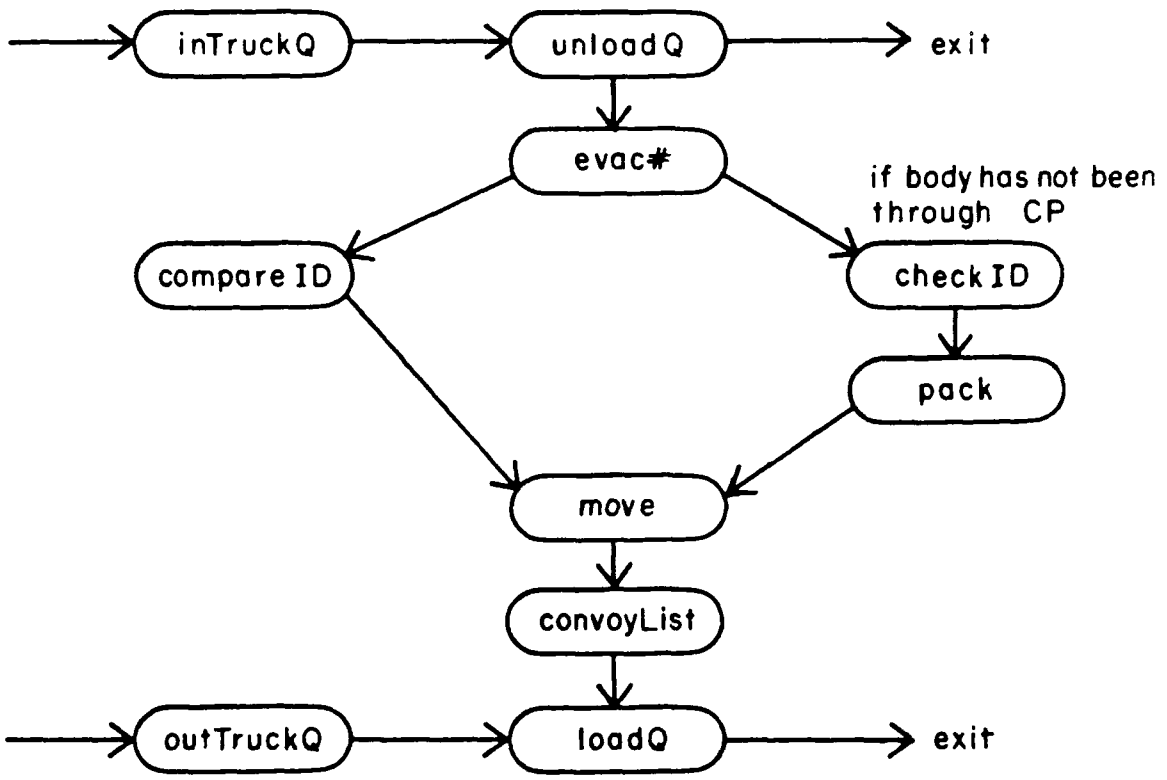


Figure 4. Intermediate Collecting Point Task Network

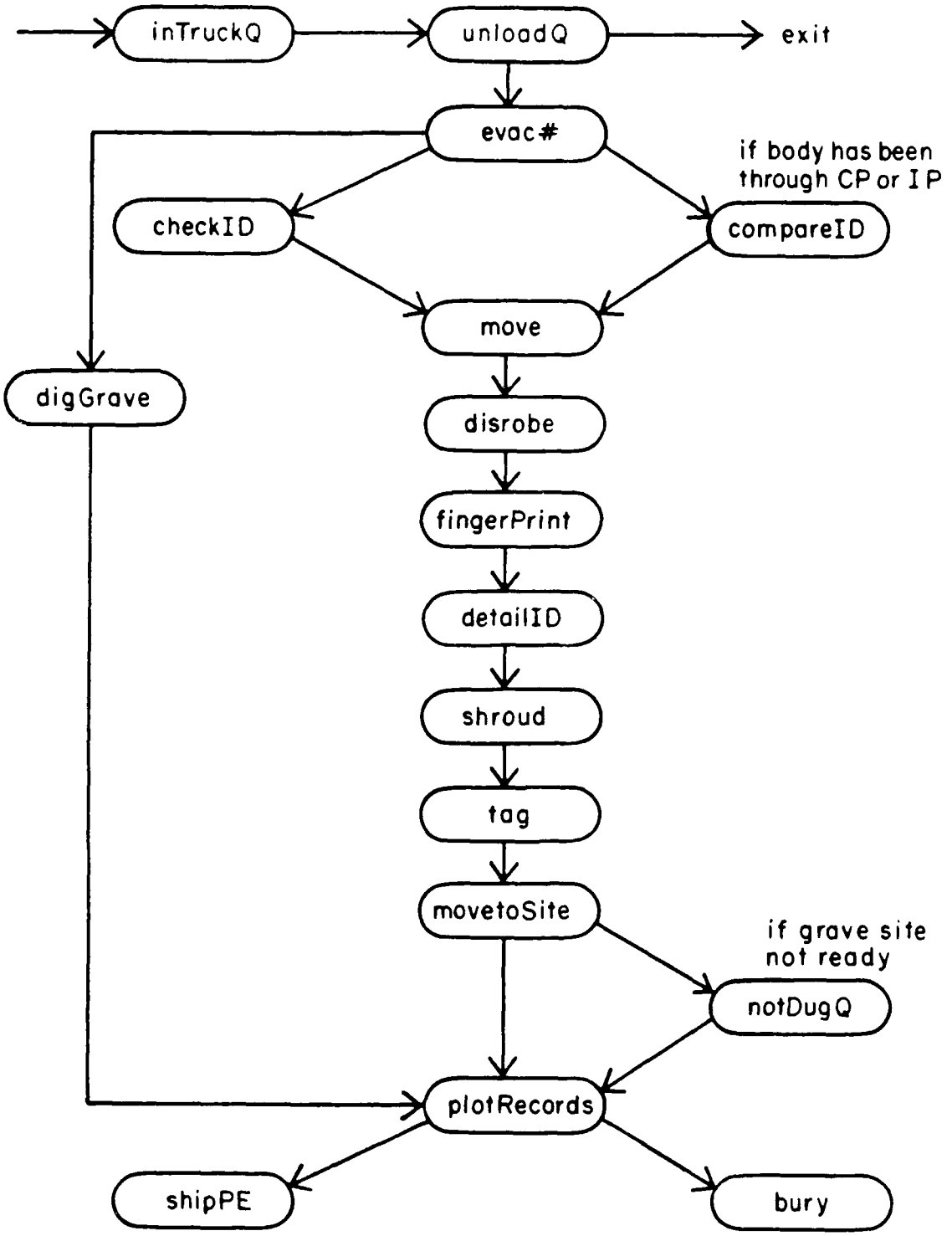


Figure 5. Temporary Cemetery Task Network

Exhibit 2. Source Code for Tasks

```
"
#       [ startUpTask
#       | max           number of workers allowed.
#       | min           number of workers required for one body.
#       | taskTime      return the average time required.
#       | prefix: echelon
#       | next
"



"---------------- If there is a truck unload it."
Class Unloader :Task

[ startUpTask
        ^self.
| max           "number of workers allowed."
        ^7.
| min           "number of workers required for one body."
        ^2.
| taskTime      "return the average time required."
        ^2.
| prefix: echelon
        ^self name: ( echelon, 'Unload' )
"----------------"
| next
        self setWorking.
        self body: (self unloadTruck: self).
]



"----------------If there is a truck load it."
Class Loader :Task
[ startUpTask
        ^self.
| prefix: echelon
        ^self name: ( echelon, 'Load' )
| max           "number of workers allowed."
        ^6.
| min           "number of workers required for one body."
        ^2.
| taskTime      "return the average time required."
        ^2.
| next
        self setWorking.
        self body:   (self loadTruck: self).
]
```

```
"---------------- Process to assign evacuation numbers. "
Class Evac :Task
[ startUpTask
        ^self.
| max             "number of workers allowed."
        ^1.
| min             "number of workers required for one body."
        ^1.
| taskTime        "return the average time required."
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Evac' )
| next
        self setWorking.
        self body:   (self procEvac: self ).
        self reSchedule: self.
]



"----------- See who this is and process his personal effects."
Class Id :Task
[ startUpTask
        ^self.
| max             "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^1.
| taskTime        "return the average time required"
        ^40.
| taskClock | c |
        c _ super taskClock.
        (self body) fingerPrint ifTrue: [ c incMin: 15 ].
        ^c.
| prefix: echelon
        ^self name: ( echelon, 'Id' )
| next
        self setWorking.
        self body:   (self procId: self).
        self reSchedule: self.
]



"------ Pack into transport bag with PE and move to load area."
Class Pack :Task
[ startUpTask
        ^self.
| max             "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
```

```
        ^2.
| taskTime      "return the average time required."
        ^10.
| prefix: echelon
        ^self name: ( echelon, 'Pack' )
| next
        self setWorking.
        self body:   (self procPack: self).
        self reSchedule: self.
]



"---------------- Add him to the convoy list."
Class Dd :Task
[ startUpTask
        ^self.
| max           "number of workers allowed."
        ^1.
| min           "number of workers required for one body."
        ^1.
| taskTime      "return the average time required."
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Dd' )
| next
        self setWorking.
        self body: (self procDd: self).
        self reSchedule: self.
]



"---------------- Process other overhead items required."
Class Misc :Task
[ startUpTask
        ^self.
| max           "number of workers allowed."
        ^999.
| min           "number of workers required for one body."
        ^1.
| taskTime      "return the average time required."
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Misc' )
| next
        self setWorking.
        self body: (self procMisc: self).
        self reSchedule: self.
]
```

VIII. Code Details

In this chapter, parts of the code will be examined in greater detail. The first part covers the details of starting the simulation. The second part covers the concept of global information in the simulation. The last part walks through the various stages in the life of a worker, including creating a worker, assigning a task, performing a task, and returning to idle status.

1. Running the Simulation

The simulation is run with two commands. The first is:

```
s <- Simulation new startUp.
```

which will make an instance of the Simulation and pass in the message startUp. This will in turn pass the message startUp to each object in the simulation that needs to be created at time zero. Each startUp will initialize the variables and the queues needed for the run, along with scheduling objects to perform operations in the future.

Each object that represents a task that consumes time is added to the event-queue. This is called 'scheduling an event'. Each object scheduled is placed on the event-queue. The event-queue is a sorted list. The records of the list are the time-to-run, which is the sort key, and the object to run.

The second command is:

```
s process.
```

This removes the first object from the event-queue and runs it. In normal operation the message 'process' returns true as long as there are objects scheduled to do tasks. Also the run can be limited by the number of events run. This is the way the simulation is run until there is an internal stop condition or 50 events have occurred:

```
[ (s process) and: [ (n<- n+1) < 50 ] ] whileTrue: []
```

Each task is run in three phases. In the first phase the task gets the resources required for the task. The second phase is the scheduling where the task is suspended for the time required to complete the task. The last phase is to pass the message 'next' to the object. The message 'next' causes the

resources to be released by the object completing a task.

2. Global Variables

One of the first things done in starting the Simulation is to make an instance of the Class GlobalData. This in turn makes an instance of the superClass UserData. Both are saved in the Class UserAccess, which is a super class of each SimulationObject. The result is two objects that are global to the Simulation. These two Classes are used to store variables that are accessed through the super-class.

Consider a simple example to count some kind of event.

```
Class UsedData
        | counter |
[ new
        counter <- 0;
| count
        counter <- counter + n
| report
        'counter = ', (counter printString) print.
]
        Class UserAccess
        | userData |
[ udSet:
        userData <- ud
| count
        userData count.
]

Class Object :UserAccess
[ next
        self count
]
        Class Main
[ main |theUd obj1 obj2|
        theUd <- UserData new.
        obj1 <- Object new udSet: theUd.
        obj2 <- Object new udSet: theUd.
        5 timesRepeat: [ obj1 next ]
        10 timesRepeat: [ obj2 next ]
        theUd report.
]
```

The result of running 'Main new main' would be to print the number of times the Class Object was passed the message 'next'. The way this works is that both of the super objects of obj1 and obj2 have the value of the variable ud set to the same instance of the Class UserData. Thus there is only one copy of the variable counter accessible from all the objects.

The Hierarchy of Classes is:

```
Probability
    Uniform
UserData
    GlobalData
UserAccess
    Identity
        CpTrucks
            IntermediatePoint
            CollectionPoint
        Truck
        GenTruck
        Body
        IdleTask
        Environment
        CollectionClass
            Worker
            Task
                DemonTask
                Move
                CkId
                Misc
                Dd
                Pack
                Id
                Evac
                Loader
                Unloader
```

Note that all of the objects are sub-Classes of UserAccess. This allows the global variables in UserData to be accessed through UserAccess from the one copy of UserData passed into each object at the time of object creation.

3. The Life of a Worker

In this last section, parts of the code relating to the life of a worker will be examined in greater detail. A listing of the complete source code is given in Appendix A, but for the reader's convenience, relevant parts will be reproduced below as needed with the discussion. The various stages in the life of a worker to be discussed are: creating a worker, assigning a task, performing a task, and returning to idle status.

a. **Start Up.** Workers are created and "live" in the class IdleTask. The scheduler takes an idle worker and assigns him to a task. When the task is over, the worker is returned to the idle state, to await reassignment to another task. When the simulation starts, the message 'startUp' is sent to the class

IdleTask, (see the previous section), and the number of workers in the collection point is passed on the message 'personnel:' as shown below.

```
Class IdleTask  :Identity
"Data"
| cp idlers timesIdle maxIdle allWorkers |
[

| startUp
        idlers          <- List new.
        allWorkers      <- List new.
        maxIdle <- 12.
        timesIdle <- 0.
        self start.
| setCP: cpoint
        cp <- cpoint

| personnel: n
        n timesRepeat: [
            allWorkers add: (
                idlers add: ( (cp create: Worker) setCP:cp )) ].
 . . .
```

As shown above, two lists of workers are generated. The workers on the 'idlers' list are removed as each worker is assigned a task. The 'allWorkers' is saved intact for listing the status of the workers. The 'startUp' message is passed from the command

```
s <- Simulation new startUp
```

discussed in the previous section.

b. **Scheduling Tasks.** After the IdleTask is created, it is scheduled by the message 'scheduleNow' in the class Collection-Point, as seen below.

```
Class CollectionPoint   :CpTrucks
 . . .
        ( ( (self iTask: (self create: IdleTask)) setCP: self
          ) personnel: pers
        ) scheduleNow.
 . . .
```

When the IdleTask is scheduled, the message 'next' is passed to the IdleTask as shown below.

```
Class IdleTask  :Identity
 . . .
| next          | rt |
        timesIdle <- timesIdle + 1.
        rt <- cp reTask: idlers.
        rt ifTrue: [ timesIdle <- 0. ].
        timesIdle >= maxIdle ifTrue: [
                cp stop.
                maxIdle <- 0.
        ]
```

Here the IdleTask will terminate the simulation (c.f. "cp stop") if there is no more work to be done. This is determined by the returned value of 'retask:'. In the CollectionPoint Class 'reTask:' tries to find work for the idle workers. The scheduler will make 12 (c.f. maxIdle <- 12) attempts to schedule idle workers.

When a worker finishes a task he looks for work. If there is no work to be done then he is idle. This worker will then add himself to the 'idlers' list by passing the IdleTask the message 'idleWorker:'. This is illustrated in the following code.

```
Class CpTrucks  :Identity
 . . .
        iTask idleWorker: worker.
 . . .

Class IdleTask  :Identity
 . . .
| idleWorker: w
        w sleeping ifFalse: [
                ( idlers isEmpty and:
                    [ maxIdle > 0 ]     ) ifTrue: [
                        self schedule:5
                ].
                w setIdle.
                idlers add: w.
        ].
]
```

It is possible to have a worker that has either worked into the night or worked a full work day of 7.5 hours. These workers are scheduled to wake up in the morning. If they are not asleep, and the 'idlers' list is empty, then the idleTask is not on the 'eventQueue'. This requires that the IdleTask be scheduled so that the workers can be given tasks. Then each worker is marked as Idle and added to the list of 'Idlers' for tasking at the scheduled time, as shown above.

From above, one sees the IdleTask being scheduled by the message 'reTask:'. Here each worker is removed from the idlers list and given to the CollectionPoint Class with the message 'taskSelect:'. If any worker is given a task, then the returned value is true. This informs the IdleTask not to terminate. Any workers that remain, after all the doable tasks are assigned workers, are idle and are passed back to the IdleTask (shown below as iTask).

```
Class CpTrucks  :Identity
 . . .
"---------------- This list of workers needs to find work."
| reTask          ": helpers"
        helpers notNil ifTrue: [ self reTask: helpers ]

| reTask: list  | l val |
        val <- false.
        l <- List new.
        [ list isEmpty ] whileFalse:
            [   l add: (list remove) ].
                                              "give them tasks."
        [ l isEmpty not and:
            [ (self taskSelect: (l remove))
        ]   ] whileTrue: [ val <- true ].
                                              "no more jobs case."
        [ l isEmpty ]whileFalse:[iTask idleWorker:(l remove)].
        ^val
 . . .

Class CollectionPoint   :CpTrucks
 . . .
| taskSelect: worker

        (self   selectIdle:     worker) ifTrue: [^true].
        (self   selectInTruck:  worker) ifTrue: [^true].
        (self   selectOutTruck: worker) ifTrue: [^true].
        (self   selectHelp:     worker) ifTrue: [^true].
        (self   selectNoWorker: worker) ifTrue: [^true].
        (self   selectBigQ:     worker) ifTrue: [^true].

        iTask idleWorker: worker.

        ^false
```

c. **Task Priorities.** The message 'taskSelect:', seen above, tries a set of schemes to give a worker a task. The order of schemes defines a priority, in that the first task found is the one assigned to the worker. The task priorities from highest to lowest are:

- end of the day
- trucks to be unloaded
- trucks to be loaded
- workers needing assistance
- tasks with no workers assigned
- tasks with a large backlog
- perform the previous task again
- random choice

The first priority is to check for night time, since work can not be performed in darkness. The next priority is to unload incoming trucks, then to load trucks. The fourth priority is to find helpers when needed. For example, some task might require two workers yet have only one worker currently assigned. Thus one helper is needed. The fifth priority is to fill tasks where no workers are assigned. Then if all tasks have workers assigned, the sixth priority is to reduce large backlogs. If there are none, then the seventh priority is to reassign the worker to the previous task. Finally, if the worker had no previous task, the last priority is to choose one at random.

Starting at the top of the list, a few of these tasks will be examined in more detail. The first scheduling priority is to check for the end of the work day. The message 'selectIdle:' checks for:

1. working over 7.5 hours in one day
2. the condition of 'lightsOut'

Current doctrine specifies a maximum of 7.5 hours per day per worker to be devoted to GRREG tasks. The condition 'lightsOut' occurs when the CollectionPoint is close to the front and it is not safe to run lights at night. If either of these are true, then the worker goes to sleep. This requires that he be scheduled for wakeup in the morning. This has been structured as follows.

```
Class CollectionPoint   :CpTrucks
 . . .
| selectIdle:   worker  |ck t h|
        ( (worker hoursWorked) >= workMax or: [self lightsOut] )
        ifTrue:[
            h <- worker todayWorkTime.
            t <- self timeIs.
            ck <- self morning: (self timeIs).
            (ck - t) < h ifTrue: [ ck <- t incHours: h ].

            worker sch: ck.
            worker setIdle.
            worker setSleeping.

            ^true
        ].
        ^false

| selectInTruck: worker
        ( (enterQ notEmpty or: [ inTruck notNil ]
          )and: [ unloadW size < unloadWorkersMax ]
        ) ifTrue: [
                self taskInTruck: worker.
                ^true
        ].
        ^false

| selectOutTruck: worker
        ( ( loadW size < loadWorkersMax
            and: [ exitQ notEmpty  or: [self outTruckLoadable] ]
          ) and: [ loadQ size >= truckMin ]
        ) ifTrue: [
                self taskOutTruck: worker.

                ^true
        ].      ^false
 . . .
```

The next two schemes for tasking are to unload and to load trucks, in that order. If the trucks are ready to be unloaded or loaded, then the messages 'taskIntruck:' and 'taskOuttruck:' are used to start the workers unloading or loading trucks respectively. Scheduling workers for trucks is more complex than most tasks since it takes helpers on the trucks to move the bodys to the tail gate, where pairs of workers can take the bodies to the first queue for assigning evacuation numbers.

The table in the comment below is used to determine what each worker is to do in order to unload trucks. For example, the first and second workers passed in to 'taskInTruck:' are stored on a list of helpers, since two workers alone cannot unload a

truck. The third worker passed in functions as one of the helpers and gets onto the truck, and the other two remain on the ground to carry bodies. Note also that the trucks must be moved up to the unloading location with the message 'deQintruck'.

```
Class CpTrucks  :Identity
 . . .
"-------------- Getting a new worker for unloading the trucks."
| taskInTruck: worker   | s w |
        worker setIdle.
        helpers add: worker.    "This worker is ready to work."
        s <- helpers size.
        self deQinTruck.    "Get truck in place for unloading."

"                                   Put helpers to work unloading.
#       number of       action for the current number of
#       helpers         unloading workers.
#                       0       3       5       7
#
#       --              ----    ----    ----    ----
#       1               -       -       -       t
#       2               -       a2      a2      t
#       3               a3      a2      a2t     t
#       4               a3      a4      a2t     t
#       5               a5      a4      a2t     t
#       6               a5      a4t     a2t     t
#       7               a7      a4t     a2t     t
#       >7              a7t     a4t     a2t     t
#
#       Key:    aN = add N workers to the task.
#               t = retask all help workers.  They are excess.
"
        (w <- unloadW size) = 0         ifTrue: [ self itw0:s ]
            ifFalse: [ w = 3            ifTrue: [ self itw3:s ]
                ifFalse: [ w = 5        ifTrue: [ self itw5:s ]
                    ifFalse: [ w = 7    ifTrue: [ self reTask ]
                        ifFalse: [ self print: [( 'error: ',
                                (w printString),' workers
                                are unloading a truck.' )]
            ]   ]   ]   ]

        "       The logic of the truck tables got to big to put
        # into one method, so cut out each column of the table.
        # For a value of w, and each range of s do the action."
| itw0: s      "(w=0) [3-4]:a3 [5-6]:a5        >6: a7t "
        s > 6 ifTrue: [ self addInTruck: 7.  self reTask ]
            ifFalse: [ s > 4 ifTrue: [ self addInTruck: 5 ]
                ifFalse: [ s > 2 ifTrue: [ self addInTruck: 3 ]
            ]   ]


| itw3:s        "(w=3) [2-3]:a2 [4-5]:a4        >5: a4t "
```

```
        s > 3 ifTrue: [ self addInTruck: 4.  self reTask ]
            ifFalse: [ s > 1
                ifTrue: [ self addInTruck: 2.
                          self reTask
                ]           ]

| itw5: s       "(w=5) >2:a2t"
        s > 1 ifTrue: [ self addInTruck: 2.
                        self reTask
                      ].

        "Add a number of workers to the job of unloading."
| addInTruck:  n          | l h |
        "n:start          2:1p    3:1p+1  4:2p    5:2p+1  7:3p+1"

                "The odd man's job is to get onto the truck."
        n odd ifTrue: [
                unloadW add: (h <- helpers remove setWorking).
                self pr:[ ((h printString),
                           ' jumps on the truck.')]
        ].

        [ n > 1 ] whileTrue:
            [   n <- n - 2.
                l <- List new.
                l add: ( helpers remove).
                l add: ( helpers remove).
                ( ( (l first) task: l create: Unloader
                  ) setCP: self
                ) scheduleNow.
                l do: [ :w | unloadW add: w ].
            ].
  . . .
```

The message 'addInTruck:', listed above, tells how many workers are to be allocated to the task of unloading a truck. If the number is odd, then the odd man is used on the truck and the rest are scheduled to take off a body with the task class Unloader.

The super class of Unloader is the class Task. This task stores: the worker doing the task; the body the task is done to; the name of the task; and the probability distribution for the task time.

The super class of Task is the class CollectionClass. In the code above, the message 'setCP:' is passed to the Unloader, and the super super class CollectionClass stores the class CollectionPoint for message passing from the task Unloader to the CollectionPoint.

```
        "Remember the Collection Point class for all workers."
Class CollectionClass   :Identity
        | cp |
[ setCP: cPoint
        cp _ cPoint
| cp
        ^cp

| nightTime
        ^cp nightTime

| lightsOut
        ^cp lightsOut

| idleTask: tObj          "-- All of these workers are now idle."
        (tObj workers) do: [ :w | cp idle: w ].
| idleWorker: w
        cp idleWorker: w.

| unloadTruck: tObj
        ^(cp unloadTruck: tObj)
| loadTruck: tObj
        ^(cp loadTruck: tObj)
| procEvac: tObj
        ^cp procEvac: tObj
| procId: tObj
        ^cp procId: tObj
 . . .

| listWorkerTask
        cp listWorkerTask
| reTask: wl
        cp reTask: wl
| procDemon
        ^cp procDemon

]

Class Task      :CollectionClass
      |  body
         workerList      "A list of the workers moving the body."
         name
         uniTime
      |
[ startUp
        body <- nil.
        uniTime <- Uniform new var20: (self taskTime).

| taskClock     | tck |         "Get random time for this task."
        tck <- uniTime taskClock.

        ^tck
```

```
| name: n                          "store the name of this task."
        ^name <- n
| printString                      "tell the name"
        ^name.
        body isNil ifTrue:  [ ^name ]
                   ifFalse:
                   [ ^( name, '(', (body printString), ')' )].

| body: b                          "Store the body."
        body <- b
| body                            "retrieve the body"
        ^body
"----------------"
| setWorking    | w |            "For all workers on this list,
                                   set them as working."
        w <- workerList first.
        [ w notNil ] whileTrue:
            [   w setWorking.
                w <- workerList next
            ].
| workers                         "retrieve the worker list."
        ^workerList.
| workers: wl           "Put workers on the worker list."
        workerList <- wl.

| reSchedule: wObj      "If there is a body then do the task."
        body notNil ifTrue: [
                wObj scheduleAfter:  (wObj taskClock)
        ] ifFalse: [
                self reTask: workerList.
        ]
]

Class Unloader :Task
[ startUpTask
        ^self.
| max             "number of workers allowed."
        ^7.
| min             "number of workers required for one body."
        ^2.
| taskTime        "return the time required."
        ^2.
| prefix: echelon
        ^self name: ( echelon, 'Unload' )
"----------------"
| next
        self setWorking.
        self body: (self unloadTruck: self).
]
```

The Unloader class message 'next' is received, and the workers on this task are marked as working, and the message

'unloadTruck:' is passed to self. This message is intercepted by the super super class CollectionClass. There the CollectionPoint class is stored as 'cp'. This allows the message to be forwarded to the CollectionPoint.

Here the trucks can be accessed so one can get a body. And one can access the queues for putting the body when a task is done. The running of a task procedure is a transaction where the object Unloader has been scheduled. The actions are:

1. put the old body, if any, on the output queue.
2. test for being idle.
3. get a new body from the truck.
4. if there are no more bodies, then retask the workers.
5. if there is a body, schedule after the time needed for the task.

These actions can be seen below.

```
Class CollectionPoint   :CpTrucks
 . . .
                        "----   Handle incoming bodies. (next)"
| unloadTruck: tObj |bdy t|

        bdy <- self     unloadl: tObj.
        bdy notNil ifTrue: [ tObj scheduleAfter:
                                        (tObj taskClock) ].

        ^ bdy.

| unloadl: tObj         |wl bdy|

        wl <- (tObj workers).
        bdy <- tObj body.
        tObj body: nil.

        self    put:    bdy     by:     wl
                on:     unloadQ msg:  'completes unloading, of'.

        bdy <- nil.

        self moveOutOverFlow:   wl.
        self deQinTruck.

        (inTruck notNil and: [(self testIdleInTruck: tObj) not])
                ifTrue: [

                bdy <- self     getBy:  wl
                                from:   inTruck
                                msg:    'starts unloading, of'.
        ].
                                        "Free the workers."
        bdy notNil ifTrue: [
                bdy enter.
        ]ifFalse:[                      "Free the workers."
                self freeReTask: unloadW.
        ].
        ^bdy
```

The code for the other priority schemes is given in the appendix, and will not be discussed.

IX. Conclusions

Object oriented languages have become popular in the 1980's. Their strengths have been in the areas of simulation, graphics, project/software management, and AI. This report restricts its scope to simulation and does not touch upon other strong points of SMALLTALK. These other fields are being actively pursued at the BRL and future reports will be concerned with additional applications of SMALLTALK.

SMALLTALK provides a natural framework for simulation studies. Programs can be easily written, tested, debugged, run, and modified. The GRREG simulation could not have been done in the time available using any other language or simulation package.

X. Summary

This paper discusses a simulation project, written in SMALLTALK, that was performed by the BRL for the Graves Registration (GRREG) Service of the US Army Quartermaster School. The purpose of the report is first to give the reader an introduction to the concept of object oriented programming, and second to show how the object oriented language SMALLTALK was used for the GRREG simulation. The paper first describes some of the features of an object oriented language, including a discussion of objects, messages, and inheritance. Next, the syntax and semantics of the object oriented language SMALLTALK are presented. There follows a synopsis of the GRREG study conducted by the BRL, and a look at how SMALLTALK was used to simulate the GRREG task network. Finally, the report steps through sections of the code to give the advanced reader a glance at how some generic actions can be programmed in SMALLTALK.

REFERENCES

1. Stefek, Mark and Bobrow, Daniel. "Object-Oriented Programming: Themes and Variations," AI Magazine, Winter 1986, pp. 40-62.

2. Dahl, O.J. and Nygaard, K. "SIMULA - An Algol Based Simulation Language," Comm. ACM, No. 9, 1966, pp. 671-678.

3. Rentsch, Tim. "Object Oriented Programming," Dept. of Computer Science, Working Paper, UCLA, n.d.

4. Goldberg, Adele and Robson, D. "Smalltalk-80: The Language and its Implementation," Addison-Wesley, 1983.

5. Fikes, R. and Kehler, T. "The Role of Frame-Based Representation in Reasoning," Comm. ACM, Vol. 28 No. 9, 1985, pp. 905-920.

6. Goldstein, I.P. and Roberts, R.B. "NUDGE, A Knowledge-Based Scheduling Program," IJCAI-1977, pp. 257-263.

7. Stefik, Mark. "An Examination of a Frame-Structured Representation System," IJCAI-1979, pp. 845-852.

8. Rees, J.A., Adams, N.I., and Meehan, J.R. "The T Manual," Yale Univ. Technical Report, Jan. 1984.

9. Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L.,Stefik, M. and Zdybel, F. "CommonLoops: Merging Common Loops and Object- Oriented Programming," ISL-85-8, Xerox PARC, Aug, 1985.

10. Cohen, A. Toni. "Data Abstraction, Data Encapsulation, and Object Oriented Programming," Dept. of Computer and Information Sciences, Working Paper, Univ. of Delaware, n.d.

11. Shankar, K.S. "Data Structures, Types, and Abstractions," Computer, April, 1980, pp. 67-77.

APPENDIX

Source Code

Introduction

This appendix contains a listing of the source code for the GRREG simulation. Exhibit A-1 lists the class hierarchy for the various subclasses used in the simulation, and Exhibit A-2 gives a brief description of those subclasses. The remainder of the appendix lists the source code. The classes are listed in alphabetic order, except for the Task subclasses, which, because of their similar structure, have been grouped and placed at the end.

Exhibit A-1. GRREG Class Hierarchy

```
Clock

Obj
     Simulation
     Probability
             Uniform
     UserData
             GlobalData
     UserAccess
             Identity
                 Environment
                 IdleTask
                 Body
                 CpTrucks
                     IntermediatePoint
                     CollectionPoint
                     TemporaryCemetery
                 RestartTools
                     SuperReStart
                         ReStart
                 CollectionClass
                     DemonTask
                     Truck
                     GenTruck
                     Worker
                     Task
                         InCover
                         Dd3x5
                         Mv2site
                         ShipPe
                         Dig
                         Plates
                         Shroud
                         DTailId
                         Fingerp
                         DisRobe
                         Move
                         CkId
                         Misc
                         Dd
                         Pack
                         Id
                         Evac
                         Loader
                         Unloader
```

Exhibit A-2. Class Descriptions

Clock	Contains the simulation clock.
Obj	The root of the simulation.
Simulation	Controls the processing of each event.
Probability	Used to access Random numbers.
Uniform	Transforms random numbers into distributions.
UserData	This super class allows storage of application dependent global data throughout the simulation. Note: there is only one UserData class in the simulation.
GlobalData	Stores application independent data, like the current simulation time and the event queue.
UserAccess	This super class contains a pointer and methods for acesss and update of the Global data throughout the simulation.
Identity	This super class contains access to application independent global data. As well as keeping reference to unique names for each object in the simulation, Identity allows access and update of the event queue. Thus all offspring of Identity are scheduleable objects.
Environment	Starts up an application dependent simulation.
IdleTask	This task Creates workers and holds them until there is work to be done.
Body	This task contains the attributes of a cadaver.
CpTrucks	This super Class of the collection points contains that portion of the Collection point processing that is in common with each collection point. This task contains the worker allocation Methods. As well as the queues for incomming and outgoing Trucks.

Class Description (cont.)

TemporaryCemetery	The Collection Points are a network of queues that allow the workload to flow from task(the offspring of the Class Task) to task. Each of the three Collection points contain the queues and the links from one task queue to another task queue.
IntermediatePoint	******
CollectionPoint	******
RestartTools	These three classes allow the definition of different processing configurations and collection-point to collection-point networks.
SuperReStart	******
ReStart	******
CollectionClass	This super class allows message passing from the offspring of Task to the collection points, which are offspring of CpTrucks.
DemonTask	This periodic task invokes output displays.
Truck	Defines the attributes of a truck.
GenTruck	Defines the rate at which bodies are retrieved from the field.
Worker	Defines the attributes of a collection point (MOS 57F) worker.
Task	This super task stores the workers and deceased for specific tasks associated with collection points. The offspring of Task are processing actions of workers upon a cadaver.
InCover	Place remains in grave and cover.
Dd3x5	Prepare internment and plot records.
Mv2site	Move remains to grave site.

Class Description (cont.)

ShipPe	Prepare and ship PE.
Dig	Dig grave site.
Plates	Prepare plates and tags and attach.
Shroud	Shroud remains.
DTailId	Perform detailed ID; consists of anatomical, dental, and/or skeletal charting, photography and comparison of evidence with records.
Fingerp	Fingerprint remains.
DisRobe	Remove clothing and examine.
Move	Move remains.
CkId	Compare remains with documentation.
Misc	Miscellaneous record keeping.
Dd	Prepare convoy list.
Pack	Place remains, documents, and PE in human remains pouch and move to holding area
Id	Check ID tags, field medical card, prepare statement of recognition, record of recovery (if necessary), inventory PE and fingerprint.
Evac	Assign an evacuation number and record.
Loader	Load remains on truck.
Unloader	Unload remains.

Class Body

```
     "---------------- Bodies ."
Class Body                :Identity
        | death finger beenToCP enterTime exitTime|
[ startUp
        enterTime <- Clock new.
        finger <- false.
        beenToCP <- 0.
        ^self
| deathAt: time
        death <- (Clock new) set: time
| deathAt
        ^death
| enter
        enterTime <- self timeIs.
| exit:cp       |delta|
        exitTime <- self timeIs.
        self print:[
                (enterTime printString),  '     ',
                (exitTime printString),   '     ',
                ((delta <- exitTime - enterTime)
                printString),  '        ',
                (self printString),     ' ###',
                (cp printString) ].
        ^delta.

| beenToCP: btc
        beenToCP <- btc.
| beenToCP
        ^beenToCP > 0
| beenToIP
        ^beenToCP > 1
| setIP
        beenToCP <- 2.
| setCP
        beenToCP <- 1.
| printString
        beenToCP > 1 ifTrue:
                [^super printString, '*' ]. " 2 "
        beenToCP > 0 ifTrue:
                [^super printString, '+' ]. " 1 "
        ^super printString.
```

```
| prefix
        ^'b'
| fingerP
        ^finger.
| fingerPrint   |t|
        finger ifFalse: [^finger <- true ].
        ^false

"Scheduling"
| next
        self pr:[ ( ( self printString),
                        ' Lays there.' )].
]
```

Class Clock

```
     Class Clock      :Magnitude
        | day hour min sec |     "hands of the clock."
[ new
        day <- 0.
        hour <- 0.
        min <- 0.
        sec <- 0.
| new: aClock
        self set: aClock

"Setting"
| set: aClock
        day  <- aClock day.
        hour <- aClock hour.
        min  <- aClock min.
        sec  <- aClock sec
| sec: aNumber
        sec <- aNumber.
        [ sec >= 60 ]   whileTrue:
                [ self incMin: 1.       sec <- sec - 60 ].
        [ sec < 0 ]     whileTrue:
                [ self decMin: 1.       sec <- sec + 60 ]
| min: aNumber
        min  <- aNumber.
        [ min >= 60 ]   whileTrue:
                [ self incHour: 1.      min <- min - 60 ].
        [ min < 0 ]     whileTrue:
                [ self decHour: 1.      min <- min + 60 ]
| hour: aNumber
        hour <- aNumber.
        [ hour >= 24 ]  whileTrue:
                [ self incDay: 1.       hour <- hour - 24 ].
        [ hour < 0 ]    whileTrue:
                [ self decDay: 1.       hour <- hour + 24 ]
| day: aNumber
        day <- aNumber

"UpDating"
| incSec: aNumber
        self sec: (sec + aNumber)
| incMin: aNumber
        self min: (min + aNumber)
| incHour: aNumber
        self hour: (hour + aNumber)
| incDay: aNumber
        day <- day + aNumber

| decSec: aNumber
        self sec: (sec - aNumber)
| decMin: aNumber
        self min: (min - aNumber)
```

```
| decHour: aNumber
        self hour: (hour - aNumber)
| decDay: aNumber
        day <- day - aNumber

| * aNum
        ^( ( ( (Clock new
               )incDay:  (day   * aNum)
             )  incHour: (hour  * aNum)
           )    i  Min:  (min   * aNum)
         )      inc3ec:  (sec   * aNum)

| + aClock
        ^( ( ( (Clock new: aClock
               )incDay:  day
             )  incHour: hour
           )    incMin:  min
         )      incSec:  sec
| - aClock
        ^( ( ( (Clock new: self
               )decDay:  (aClock day)
             )  decHour: (aClock hour)
           )    decMin:  (aClock min)
         )      decSec:  (aClock sec)

"Access"
| day
        ^day.
| hour
        ^hour.
| min
        ^min.
| sec
        ^sec.

| printHM        | hm s |
        hm <- (hour*100) + min.
        hm < 1000       ifTrue: [
            hm < 100    ifTrue: [
                hm < 10 ifTrue:
                          [s <- '000', (hm printString)]
                        ifFalse:
                          [s <- '00',(hm printString)]
                        ifFalse:
                          [s <- '0', (hm printString)]
                        ifFalse:
                          [s <-  (hm printString) ].
        ^s

| printString
        sec = 0 ifTrue: [ ^ (day printString),
                            ':',  self printHM ].
        ^ (day printString),  ':',  self printHM,
```

```
                        ':',  (sec printString)

| = aClock
        ^(      ( day  = aClock day  )
              &  ( hour = aClock hour )
              &  ( min  = aClock min  )
              &  ( sec  = aClock sec  ) )

"NOTE: Combining some of the parts
       makes comparisons faster."

| > aClock      | left right |
        ( (left <- self highPart) >
                (right <- aClock highPart) )
          ifTrue:  [ ^true ]
          ifFalse: [ ( left = right )
                        ifTrue:
                        [ ( sec > (aClock sec) )
                            ifTrue: [ ^true ]
                                ]
                   ].
        ^ false.

| <= aClock
        ^(self > aClock) not

| >= aClock     | z |
        ^(self < aClock) not.

| < aClock
        ^(aClock > self).

"Private"
| highPart
        ^ ( min + (hour * 100) + (day  * 10000) )
]
```

Class CollectionClass

```
     "Remember the Collection Point class for all workers."

Class CollectionClass   :Identity
        | cp |
[ setCP: cPoint
        cp _ cPoint
| cp
        ^cp

| nightTime
        ^cp nightTime

| lightsOut
        ^cp lightsOut

| idleTask: tObj
        "---- All of these workers are now idle."
        (tObj workers) do: [ :w | cp idle: w ].
| idleWorker: w
        cp idleWorker: w.

| unloadTruck: tObj
        ^(cp unloadTruck: tObj)
| loadTruck: tObj
        ^(cp loadTruck: tObj)
| procEvac: tObj
        ^cp procEvac: tObj
| procId: tObj
        ^cp procId: tObj
| procPack: tObj
        ^cp procPack: tObj
| procCkId: tObj
        ^cp procCkId: tObj
| procMove: tObj
        ^cp procMove: tObj
| procDd: tObj
        ^cp procDd: tObj
| procMisc: tObj
        ^cp procMisc: tObj

| procDisRobe: tObj
        ^cp procDisRobe: tObj
| procFingerP: tObj
        ^cp procFingerP: tObj
| procDTailId: tObj
        ^cp procDTailId: tObj
| procShroud: tObj
        ^cp procShroud: tObj
| procPlates: tObj
```

```
        ^cp procPlates: tObj
| procDig: tObj
        ^cp procDig: tObj
| procShipPe: tObj
        ^cp procShipPe: tObj
| procMv2site: tObj
        ^cp procMv2site: tObj
| procDd3x5: tObj
        ^cp procDd3x5: tObj
| procInCover: tObj
        ^cp procInCover: tObj

| listWorkerTask
        cp listWorkerTask
| reTask: wl
        cp reTask: wl
| procDemon
        ^cp procDemon

]
```

Class CollectionPoint

```
     "Brigade Level"

Class CollectionPoint   :CpTrucks
"Data"|
        evacW idW packW ddW miscW  "Workers doing tasks"
        evacQ idQ packQ ddQ miscQ  "Places to put bodys."
              loadQ
        taskList                   "a list of triples
                                   (see makeTaskList)."
        battleZone                 "Lights out at night."
"Methods"|
[ prefix: echelon       | ch |
 ['              CP | echelon = ', echelon ]display.
        ch <- echelon at:1.
        ch == $x ifTrue: [
            ^(echelon copyFrom: 2 to: (echelon size)), 'c'
        ] ifFalse: [
            ch == $t ifTrue: [  ^echelon, 'c' ]
        ].
        ^'c'

| startUp
 ['     CP | startUp    ', (self printString) ]display.
        super startTrucks.
        self startQueues.

        0 timesRepeat: [ self addBodyOn: evacQ ].
        0 timesRepeat: [ self addBodyOn: idQ ].
        0 timesRepeat: [ self addBodyOn: packQ ].
        0 timesRepeat: [ self addBodyOn: ddQ ].
        0 timesRepeat: [ self addBodyOn: loadQ ].

        self taskList: (taskList <- self makeTaskList).
        battleZone <- true.

| reStart: blist |tk bdys|
        (bdys <- blist first) = 'p' ifFalse: [  "letter"
            [ 'warning error: CP letter = ',
                (bdys printString), '.' ]display.
        ].
        bdys <- blist next.     "number of bodies"

        tk <-   ( ( ( ( (self create: GenTruck) setCP: self
                      ) queue: (self enterQ)
                ) fill: (self truckMin) to: (self truckFull)
                  ) bdys: bdys
                ) schedule: 1.

        ( ( ( ( (self create: GenTruck) setCP: self
                )queue: (self exitQ)
            ) rate: (tk rate)
```

```
          ) trucks: (bdys / 12 roundTo:1)
        ) schedule: 30.


| deceased: list  | foo pers bdys time tk  tkRate |
        [ 'CP list=', (list printString) ]display.

        (foo <- list remove) = 'C' ifFalse: [   "letter"
            [ 'warning error: CP letter = ',
                (foo printString), '.'  ]display.
        ].

        pers <- list remove.
        bdys <- list remove.
        time <- list remove.

        tk <- (((((self create: GenTruck) setCP: self )
                queue: (self enterQ) )
                fill: (self truckMin)
                to: (self truckFull)  )
                bdys: bdys  )
                schedule: 0.

        ( ( ( ( ( (self create: GenTruck) setCP: self
                )queue: (self exitQ)
              ) timeTo: time
            ) rate: (tk rate)
          ) trucks: (bdys / 12 roundTo:1)
        ) schedule: 0.

        ( ( (self iTask: (self create: IdleTask))
                setCP: self  ) personal: pers
                        ) scheduleNow.

        ( (self demon: (self create: DemonTask))
                setCP: self ) scheduleNow.

| makeTaskList  | l |
        l <- List new.
        l add: ( self makeTL: evacQ with: evacW
                        task: Evac ).
        l add: ( self makeTL: idQ   with: idW
                        task: Id  ).
        l add: ( self makeTL: packQ with: packW
                        task: Pack ).
        l add: ( self makeTL: ddQ   with: ddW
                        task: Dd ).
        l add: ( self makeTL: miscQ with: miscW
                        taskN: Misc ).
        ^l

"make the queues for the Collection Point."
| startQueues
```

```
        evacQ   <- self unloadQ.
        idQ     <- List new.
        packQ   <- List new.
        ddQ     <- List new.
        miscQ   <- List new.
        loadQ   <- self loadQ.
                                        "Workers"
        evacW   <- List new.
        idW     <- List new.
        packW   <- List new.
        ddW     <- List new.
        miscW   <- List new.

| lightsOut
        battleZone ifTrue: [    ^self nightTime ].
        ^false.

"----------------
#     NEXT
"

| procEvac: tObj | wkr bdy |
        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy  by: wkr
                on:  idQ  msg:
                        'completes Evac form DD1077, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   evacQ
                        msg:    'assigns Evac # to'.
        ].
        self free: tObj when: bdy from: evacW.
        ^bdy.


| procId: tObj          | wkr bdy |
        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy  by: wkr
                on: packQ msg:
                        'completes Id and DD forms, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   idQ
```

```
                    msg:    'starts to Id'.
        ].
        self free: tObj when: bdy from: idW.

        ^bdy.

| procPack: tObj        | wkr bdy |
        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  ddQ        msg: 'finishes Moving, of '.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   packQ
                        msg:    'start to Pack and Move'.
        ].
        self free: tObj when: bdy from: packW.

        ^bdy.

| procDd: tObj  | wkr bdy |
        wkr <- tObj workers first.
        (bdy <- tObj body) notNil ifTrue: [ bdy setCP ].
        tObj body: nil.

        self put: bdy   by: wkr on:  loadQ
                        msg:    'completes DD175, for'.

        self    put: bdy   by: wkr  on: miscQ.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   ddQ
                    msg: 'starts to fill out DD175 for'.

        ].
        self free: tObj when: bdy from: ddW.

        ^bdy.

| procMisc: tObj        | wkr bdy |
        wkr <- tObj workers first.
        tObj body: nil.


        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil
        ]ifFalse: [
```

```
            bdy <- self getBy:  wkr     from:   miscQ
                   msg: 'does the miscellaneous task.'.
        ].
        self free: tObj when: bdy from: miscW.

        ^bdy.

    | list
        self listInTruck.
        self listTask: taskList.
        self listOutTruck.

    | listWorkerTask
        self listInTruck.
        self listTask: taskList.
        self listOutTruck.
        self reTask.
    ]
```

Class CpTrucks

```
     Class CpTrucks  :Identity
"Data"|
        "Unprocessed/Processed Bodies in trucks"
        enterQ exitQ
        "The curent unloading/loading truck."
        inTruck outTruck
        "Lists of workers doing tasks."
        unloadW loadW
        "Places to put bodies."
        unloadQ loadQ
        "Through put list"
        goneQ
        "Variables (see startUp)"
        unloadWorkersMax loadWorkersMax
        truckFull truckMin pullUpTime maxBodys
        "The next stop for bodies."
        truckNext fowarding
        workMax
        "Hooks to sub classes."
        iTask   demon
        "A list of triples (see makeTaskList)."
        taskList
        "Those that are waiting to do a task,"
        helpers
        "and the task the worker is waiting for."
        helpT
        timeTo
        "Used to count the bodies at the CP."
        countofbdys
        "Report on queues information."
        qcnt    qname

"Methods"|
[ startTrucks
        unloadWorkersMax<- 7.
                "Max. number of workers for unloading."
        loadWorkersMax  <- 6.
                "       ''      ''      ''    loading."
        truckFull       <- 24.
                "Max. number of bodies in a truck."
        truckMin        <- 12.
                "Min. number of bodies to load."

        timeTo          <- 5.   "default time."

        pullUpTime <- 1.
                "Min.s to move truck up to unloading loc."
        maxBodys <- 800.
                "CP body overflow threshold."
        fowarding <- false.     "under threshold."
```

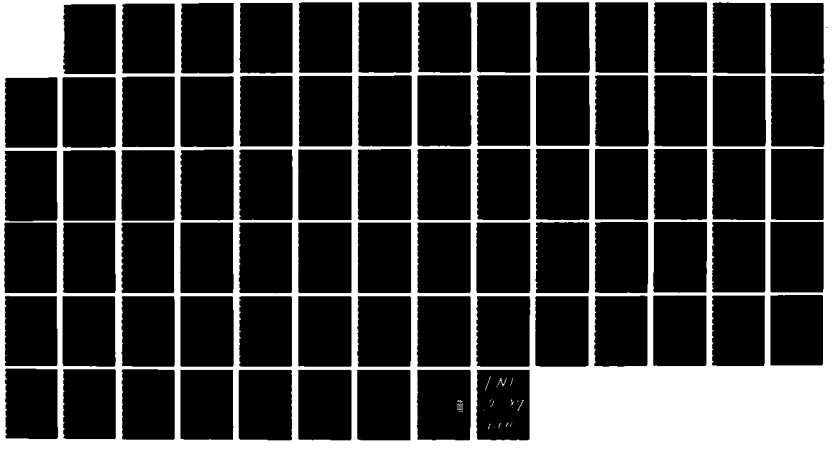
```
        workMax <- 7.5.     "7.5 hours of work per day."

        enterQ  <- List new.            "Truck Queues"
        exitQ   <- List new.
        inTruck <- outTruck <- nil.
                        "Truck currently being unloaded."
        unloadQ <- List new.    "Bodies not in trucks."
        loadQ   <- List new.
        goneQ   <- List new.

        unloadW <- List new.            "Workers"
        loadW   <- List new.
        helpers <- List new.
| timeTo: t
        timeTo <- t.
| enterQ
        ^enterQ.
| exitQ
        ^exitQ.
| truckMin
        ^truckMin
| truckFull
        ^truckFull
| truckNext: q          "The next place to take trucks."
        truckNext <- q

| stop
        demon stop
| idleWorker: w
        iTask idleWorker:w.

| demon: d
        ^demon <- d
| demon
        ^demon
| taskList: tl
        ^taskList <- tl.

| iTask: it
        ^iTask <- it
| iTask
        ^iTask
| unloadQ
        ^unloadQ
| loadQ
        ^loadQ


"----------------
#       TASK Assignments
"

| taskSelect: worker
```

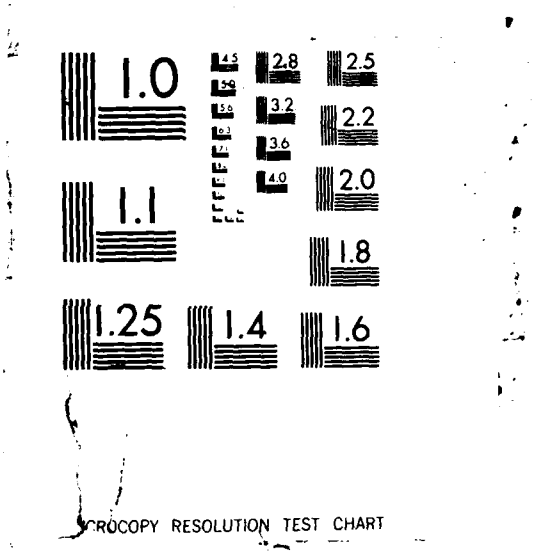
```
        worker sleeping ifTrue: [[ (worker printString),
                        ' is Sleeping. ']display.
            ^true
        ].
        (self   selectIdle:     worker) ifTrue: [^true].
        (self   selectInTruck:  worker) ifTrue: [^true].
        (self   selectOutTruck: worker) ifTrue: [^true].
        (self   selectHelp:     worker) ifTrue: [^true].
        (self   selectNoWorker: worker) ifTrue: [^true].
        (self   selectBigQ:     worker) ifTrue: [^true].

        iTask idleWorker: worker.
        ^false


| selectIdle:   worker  |ck t h|
        ( (worker hoursWorked) >= workMax
            or: [self lightsOut] )ifTrue:[
            h <- worker todayWorkTime.
            t <- self timeIs.
            ck <- self morring: (self timeIs).
            (ck - t) < h ifTrue: [ ck <- t incHour: h ].

            self print: [( (worker printString),
                        ' takes a nap, until ',
                        (ck   printString), '.' )].
            worker sch: ck.
            worker setIdle.
            worker setSleeping.
            ^true
        ].
        ^false

| testInTruck
        ( (enterQ notEmpty or: [ inTruck notNil ]
          )and: [ unloadW size < unloadWorkersMax ]
        ) ifTrue: [^true].
        ^false


| selectInTruck: worker
        self testInTruck ifTrue: [
                self taskInTruck: worker.
                ^true
        ].      ^false

| selectOutTruck: worker
        ( ( loadW size < loadWorkersMax
            and: [ exitQ notEmpty
             or: [self outTruckLoadable] ]
          ) and: [ loadQ size >= truckMin ]
        ) ifTrue: [
                self taskOutTruck: worker.
```

```
                ^true
        ].      ^false

| selectHelp: worker
        helpT notNil    ifTrue: [
                self startHelp: worker.
                ^true
        ].      ^false

| selectNoWorker: worker        | taskl |
        (taskl <- self noWorkerTask) notNil
                ifTrue: [
                self start: taskl with: worker.
                ^true
        ].      ^false

| selectBigQ: worker            | taskl |
        (taskl <- self bigestQ) notNil
                ifTrue: [
                self start: taskl with: worker.
                ^true
        ].      ^false


"----------- This list of workers needs to find work."
| reTask        ": helpers"
        helpers notNil ifTrue: [ self reTask: helpers ]

| reTask: list  | l val |
        val <- false.
        l <- List new.
        [ list isEmpty ] whileFalse:
            [   l add: (list remove) ].
                                "give them tasks."
        [ l isEmpty not and:
            [ (self taskSelect: (l remove))
        ]   ] whileTrue: [ val <- true ].
                                "no more jobs case."
        [ l isEmpty ] whileFalse:
                [ iTask idleWorker: (l remove) ].
        ^val


"----------------Over worker/night testing."
| testIdle: tObj |val|
        val <- false.
        tObj workers do:[ :w |  (self selectIdle: w)
                        ifTrue: [val <- true] ].
        val ifFalse:[ (self testInTruck)
                        ifTrue: [val <- true] ].
        ^val.

| testIdleInTruck: tObj |val|
```

AD-A175 198 SIMULATION USING SMALLTALK(U) ARMY BALLISTIC RESEARCH LAB ABERDEEN PROVING GROUND MD R A HELFMAN ET AL. OCT 86 BRL-TR-2764

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
CROCOPY RESOLUTION TEST CHART

```
        val <- false.
        tObj workers do:[ :w |  (self selectIdle: w)
                        ifTrue: [val <- true] ].
        ^val.

| nightTime      |h|      "from 8pm to 6am is nightTime."
        h <- self timeIs hour.
        ( h <= 6 or: [ h >= 20 ] ) ifTrue:
                        [" h >= 20 "
                ^ true
        ].
        ^false

"---------------- set up functions"
| makeTL: q  with: w  task:   t | l |
        l <- List new.
        l add: q.
        l add: w.
        l add: t.
        l add: true.
        ^l

| makeTL: q  with: w  taskN:   t | l |
        l <- List new.
        l add: q.
        l add: w.
        l add: t.
        l add: false.   "no Queue reports"
        ^l

"----------------       debug body generation."
| addBodyOn: q  | bdy time |
        time <- (Clock new) set: (self timeIs).
        bdy <- (self create: Body) deathAt: time.

        bdy fingerPrint.
        bdy setCP.

        q add: time
          put: bdy

"-------- Getting a new worker for unloading the trucks."
| taskInTruck: worker   | s w |
        worker setIdle.
        helpers add: worker.
                "This worker is ready to work."
        s <- helpers size.
        self deQinTruck.
                "Get truck in place for unloading."

"                       Put helpers to work unloading.
#       number of       action for the current number
#                       of unloading workers.
```

```
#       helpers         0       3       5       7
#       --              ----    ----    ----    ----
#       1               -       -       -       t
#       2               -       a2      a2      t
#       3               a3      a2      a2t     t
#       4               a3      a4      a2t     t
#       5               a5      a4      a2t     t
#       6               a5      a4t     a2t     t
#       7               a7      a4t     a2t     t
#       >7              a7t     a4t     a2t     t
#
#       Key:    aN = add N workers to the task.
#               t = retask all help workers.
#                               They are excess.
"
        (w <- unloadW size) = 0  ifTrue: [ self itw0:s ]
            ifFalse: [ w = 3     ifTrue: [ self itw3:s ]
                ifFalse: [ w = 5 ifTrue: [ self itw5:s ]
                    ifFalse: [ w = 7
                                   ifTrue: [ self reTask ]
                    ifFalse: [ self print: [( 'error: ',
                                   (w printString),' workers
                                   are unloading a truck.' )]
                ]   ]   ]   ]


  "The logic of the truck tables got too big to put
  # into one method, so I cut out each column of the
  # table.  For a value of w, and each range of s
  # do the action.
  "

| itw0: s       "(w=0) [3-4]:a3 [5-6]:a5        >6: a7t "
        s > 6 ifTrue: [ self addInTruck: 7.  self reTask ]
            ifFalse: [ s > 4 ifTrue: [ self addInTruck: 5 ]
            ifFalse: [ s > 2 ifTrue: [ self addInTruck: 3 ]
            ]   ]


| itw3:s        "(w=3) [2-3]:a2 [4-5]:a4        >5: a4t "
        s > 3 ifTrue: [ self addInTruck: 4.  self reTask ]
            ifFalse: [ s > 1
                ifTrue: [ self addInTruck: 2.
                          self reTask
            ]           ]

| itw5: s       "(w=5) >2:a2t"
        s > 1 ifTrue: [ self addInTruck: 2.
                        self reTask
                    ].


"------------ Getting a new worker for loading a truck."
```

```
| taskOutTruck: worker | s w |
        worker setIdle.
        helpers add: worker. "This worker is ready to work."
        s <- helpers size.
        self deQoutTruck.  "Get truck in place for loading."
"Put helpers to work.
#       number of       action for the current number
#                       of unloading workers.
#       helpers =s,  w= 0       2       4       6
#       --              ----    ----    ----    ----
#       1               -       -       -       t
#       2               -       -       a2      t
#       3               -       -       a2t     t
#       4               a4      a4      a2t     t
#       5               a4      a4t     a2t     t
#       6               a6      a4t     a2t     t
#       >6              a6t     a4t     a2t     t
#"
        w <- loadW size.
        (w <- loadW size) = 0
                ifTrue: [ self otw0:s ]
                ifFalse: [ w = 2
                ifTrue: [ self otw2:s ]
                ifFalse: [ w = 4
                ifTrue: [ self otw4:s ]
                    ifFalse: [ w = 6
                        ifTrue: [ self reTask ]
                        ifFalse: [ self print:
                                [( 'error: ',
                                (w printString),
                                ' workers are
                                loading a truck.')]
            ]   ]   ]   ].

| otw0:s
        s > 5 ifTrue: [ self addOutTruck: 6.  self reTask ]
            ifFalse: [ s > 3 ifTrue: [ self addOutTruck: 4]]

| otw2:s
        s > 3 ifTrue: [ self addOutTruck: 4.  self reTask ]

| otw4:s
        s > 1 ifTrue: [ self addOutTruck: 2.  self reTask ]


"----------------
#       Return the first taskClass with bodies and
#       no workers allocated.
"
| noWorkerTask | tl val |
        val <- nil.
        (tl <- taskList first) notNil ifTrue:
                [ val <- self nwt: tl ].
```

```
        ^val.

| nwt: tItem     |que wkr tsk|
        tItem isNil ifTrue: [ ^nil ].
        que <- tItem first.
        wkr <- tItem next.
        tsk <- tItem next.

        ( que isEmpty not and:[ wkr isEmpty ] )
                ifTrue: [ ^tItem ].
        ^self nwt: taskList next.

"---------------Return the taskClass with the largest
        back log.  With workers working as a factor."
| bigestQ        | tl val |
        val <- self     bigQ: nil
                        size: 1
                        task: taskList first.
        ^val.


| wkrBodys:wkr  |qw|
        qw <- 0.
        wkr do: [ :w| (w taskObj body) notNil
                ifTrue: [ qw <- qw + 1 ] ].
        ^qw

| bigQ: bItem   size:   bSize   task:   tItem
        | que wkr tsk   qs qw t tmin tmax ws    nSize |
        tItem isNil ifTrue: [ ^bItem ].
        que <- tItem first.
        wkr <- tItem next.
        tsk <- tItem next.
        que isEmpty not ifTrue:
        [   qs <- que size.

            t <- tsk new.
            tmin <- t min.
            tmax <- t max.
            ws <- wkr size.
            qs <- qs + ( (self wkrBodys: wkr) / tmin ).

            ws < tmax ifTrue:
            [   wkr isEmpty ifTrue:
                [self print:[ ('should not happen.
                                tItem = ',
                                (tItem printString) )].
                        nSize <- qs.
                ]ifFalse:[
                        nSize <- (qs * tmin) / ws.
                ].

                nSize > bSize ifTrue:   [ bItem <- tItem.
```

```
                                                bSize <- nSize.
        ]   ]                                  ].
        ^self   bigQ: bItem
                size: bSize
                task: taskList next.


"----------------
#       Start a worker doing a task.
"
| start: taskl  with:  worker   | l wkr tsk min |
        taskl first.
        wkr <- taskl next.
        tsk <- taskl next.
        min <- tsk new min.
        l <- List new.
        min = 1 ifTrue:
        [       wkr add: worker.
                l <- List new.
                l add: worker.
                ( ( worker task: l      create: tsk
                  ) setCP: self
                ) scheduleNow.
        ] ifFalse: [
                worker setIdle.
                helpers add: worker.

                helpT <- taskl.
        ].

| startHelp: worker             | l wkr tsk min w |
        helpT first.
        wkr <- helpT next.
        tsk <- helpT next.
        min <- tsk new min.
        helpers add: worker.
        helpers size >= min ifTrue:
        [       l <- List new.
                min timesRepeat:
                [       l add: (w <- helpers remove).
                        wkr add: w.
                ].
                ( ( worker task: l create: tsk
                  ) setCP: self
                ) scheduleNow.
        ].
        helpT <- nil.
        self reTask.

        "Add a number of workers to the job of unloading."
| addInTruck:  n        | l h |
        "n:sta..   2:1p 3:1p+1  4:2p    5:2p+1  7:3p+1"
        "The Odd man's job is to get onto the truck."
```

```
        n odd ifTrue: [ unloadW
                add: (h <- helpers remove setWorking).
                self print:[ ((h printString),
                        ' jumps on the truck.')]
        ].

        [ n > 1 ] whileTrue:
            [   n <- n - 2.
                l <- List new.
                l add: ( helpers remove).
                l add: ( helpers remove).
                ( ( (l first) task: l create: Unloader
                  ) setCP: self
                ) scheduleNow.
                l do: [ :w | unloadW add: w ].
            ].

        "Add a number of workers to the job of loading."
| addOutTruck: nW       | l |
        "nW:start       2:1p    4:1p+2  6:2p+2"

        l <- List new.
        nW > 2 ifTrue:          "Two men get on the truck."
                [ 2 timesRepeat:
                    [   nW <- nW - 1.
                        l add: (helpers remove setWorking).
                ]   ].
        l notEmpty ifTrue: [
                self print:[ ((l printString),
                        ' jump on the truck.')].
                l do: [ :w | loadW add: w ].
        ].

        [ nW > 0 ] whileTrue:
            [   l <- List new.
                nW <- nW - 2.
                l add: ( helpers remove).
                l add: ( helpers remove).
                ( ( (l first) task: l create: Loader
                  ) setCP: self
                ) schedule: l.
                l do: [ :w | loadW add: w ].
            ].

"----------------
#     NEXT
"
        "----   Handle Incomming bodies. (next)"
| unloadTruck: tObj |bdy t|
        bdy <- self     unload1: tObj.

        bdy notNil ifTrue:
                [ tObj scheduleAfter:(tObj taskClock)].
```

```
        ^ bdy.

| unload1: tObj          |wl bdy|

        wl <- (tObj workers).
        bdy <- tObj body.
        tObj body: nil.

        self    put:    bdy     by:     wl
                on:     unloadQ
                msg:    'completes unloading, of'.

        bdy <- nil.

        self moveOutOverFlow:   wl.
        self deQinTruck.
        (inTruck notNil and:
                [(self testIdleInTruck: tObj) not])
                ifTrue: [ bdy <- self
                        getBy:  wl
                        from:   inTruck
                        msg:    'starts unloading, of'.
        ].
                                "Free the workers."
        bdy notNil ifTrue: [
                bdy enter.
        ]ifFalse:[              "Free the workers."
                self freeReTask: unloadW.
        ].

        ^bdy

"               ---     Are there too many trucks?"
| moveOutOverFlow: worker       | trk |
        fowarding ifTrue: [
            (trk <- enterQ remove) notNil ifTrue: [
                self foward: trk  by: worker
        ]   ].

| foward: trk by: worker
 ( '| foward: ', (trk printString), '   by: ',
                (worker printString) )print.
        self print:[ (  (trk printString),
                        ' is fowarded by ',
                        (worker printString), ',')].
        self print:[ ( '        to drive off
                                to the next station.' )].
        trk leave: truckNext timeTo: timeTo.


        "----   Handle  Outgoing bodies. (next)"
| loadTruck: tObj               | bdy t |
        bdy <- self     ld1: tObj.
```

```
        bdy notNil ifTrue:
                [ tObj scheduleAfter: (tObj taskClock) ].

        ^ bdy.

| ldl: tObj             |wl bdy|

        wl <- (tObj workers).
        bdy <- (tObj body).
        tObj body: nil.
        self    put:    bdy             by:     wl
                on:     outTruck
                msg:    'completes loading, of'.

        self    put: bdy   by: wl  on: goneQ.

        bdy <- nil.

        self deQoutTruck.
        ( self outTruckLoadable
                and: [(self testIdle: tObj) not] )
                ifTrue: [
                bdy <-  self
                        getBy:  wl
                        from:   loadQ
                        msg:    'starts loading, of'
        ].

        bdy notNil ifTrue: [demon body: bdy
                delta: (bdy exit: self).
        ]ifFalse:[              "Free the workers."
                self freeReTask: loadW
        ].

        ^bdy


        "Move the truck to the unloading area."
| deQinTruck    |pt|
        self moveOutOverFlow: nil.
        inTruck isNil ifTrue: [
                pt <- enterQ remove.
                inTruck <- pt.
        ]ifFalse:[
            inTruck isEmpty ifTrue: [
                self print:[( (inTruck printString),
                                ' exits.' )].
                inTruck leave: nil timeTo: timeTo.
                inTruck <- nil.
                self deQinTruck.
            ]
        ].
```

```
        "Move the truck to the loading area."
| deQoutTruck   |pt|
        self moveOutOverFlow: nil.
        outTruck isNil ifTrue: [
            (pt <- exitQ remove) notNil ifTrue: [
                outTruck <- pt.
        ]   ]ifFalse:[
            (outTruck size >= truckFull or: [
                loadQ isEmpty and:
                                [ self loadingBodys not ] ]
            ) ifTrue: [
                    self print:[( (outTruck printString),
                                        ' exits.' )].
                    outTruck leave: truckNext
                            timeTo: timeTo.
                    outTruck <- nil.
                    self deQoutTruck.
        ]   ].

                "Are there Bodies being loaded."
| loadingBodys
        ^ loadW inject: false into: [ :p :w |
                p or: [w body notNil]
          ]


| outTruckLoadable      |bdyInTruck|
        (outTruck notNil and:[ loadQ notEmpty ] )
            ifTrue: [
            bdyInTruck <- outTruck size.
            bdyInTruck < (truckFull - 1) ifTrue:[
                ^true
            ]ifFalse:[
                (bdyInTruck < truckFull
                        and: [ self loadingBodys not]
                ) ifTrue:[
                        ^true
        ]   ]   ].
        ^false

| procDemon
        ^nil.

| freeReTask: wl         | l |
        l <- List new.
        wl do: [ :w | w body isNil ifTrue: [
                        l add: (wl remove: w)
        ]   ].
        self reTask: l


| put: body      by: worker      on: outQ
        self    put:    body    by:     worker
```

```
                on:     outQ    msg:    nil

| put: body     on: outQ
       self     put:    body    by:     nil
                on:     outQ    msg:    nil

| getBy: worker   from: inQ
       ^ self  getBy:  worker  from:   inQ
                msg:    nil

| put:  body  by: worker  on: outQ  msg: msg
       ( body notNil ) ifTrue: [
           ( msg notNil ) ifTrue: [
               self msg: msg wkr: worker bdy: body.
           ].
           outQ         add: (body deathAt)
                        put: body
         ].
                "aquire a body from the inQ"
| getBy: worker  from: inQ  msg: msg    | body |
       ( (body <- inQ remove) notNil ) ifTrue: [
           self msg: msg wkr: worker bdy: body.
       ].
       ^ body


| msg: msg wkr: worker bdy: body
       ( msg notNil ) ifTrue: [
           self pr:[ ( (worker printString),    ' ',
                       msg,     ' ',
                       (body printString),      '.' )]
       ].


| free: tObj when: bdy from: wkrs
       bdy isNil ifTrue: [
           (tObj workers) do: [ :w |            "reTask"
                wkrs remove: w
       ]   ].

| listTrucks | tl que wkr tsk b |
       ['--------------------------------'] display.
       ['      idlers=', (iTask idlers printString)
                                        ]display.
       ['      inTrk=',  (inTruck      printString),
        '      enterQ=', (enterQ       printString)
                                        ]display.

       ['      unloadW=', (unloadW     printString)
                                        ]display.
       ['      helpers=', (helpers     printString),
        '      helpT=',   (helpT       printString)
                                        ]display.
```

```
        ['      loadQ=', (loadQ printString) ]display.
        ['      loadW=', (loadW printString) ]display.

        ['      outTrk=',  (outTruck    printString),
         '      exitQ=',   (exitQ       printString)
                                        ]display.
        ['--------------------------------'] display.

"---------------- Demon Queue listing messages."
| countIs
        ^countofbdys.

| countBdy: b
        countofbdys <-  countofbdys + 1.
        ^1

| countQue: que          | cnt |
        countofbdys <-  countofbdys + (cnt <- que size).
        ^cnt

| countWrks: wl          | b cnt bset |
        cnt <- 0.
        bset <- Set new.
        wl do: [ :wkr |
            (b <- wkr body) notNil ifTrue: [
                (bset includes: b) ifFalse:[
                    bset add: b.
                    cnt <- cnt + 1.
                    countofbdys <- countofbdys + 1.
        ]   ]   ].
        ^ cnt
| countTruckQ: trkQ      | cnt |
        cnt <- 0.
        trkQ do: [ :trk | cnt <- cnt +
                                (self countTruck: trk) ].
        ^ cnt

| countTruck: trk        | cnt |
        trk notNil ifTrue: [ ^trk size ].
        ^ 0

| qcntReport
        (' @  Queues:', qname,
                '@@@', (self printString)       )print.
        (' @   ', (self timeIs printString),    qcnt,
                '$$$'  (self printString)       )print.

| append: tsk to: cnt    | t |
        qcnt  <- qcnt,  (cnt printString), '    '.
        qname <- qname, tsk,               '    '.

| listTask: taskL | tl que wkr tsk cnt reportP |
```

```
        tl <- taskL first.
        [tl notNil] whileTrue: [
            que <- tl first.
            wkr <- tl next.
            tsk <- tl next.
            reportP <- tl next.
            reportP ifTrue: [
"               ('  @    ', (tsk printString),
                         '       W=', (wkr printString),
                         '  Q=', (que printString) )print.
"
                cnt <-          (self countQue: que).
                cnt <- cnt +    (self countWrks: wkr).
                self append: (tsk  printString) to: cnt.
            ].

            (   ' @    ', (tsk printString),
                '       W=', (wkr printString),
                '       Q=', (que size printString)) print.

            tl <- taskL next.
        ].

| listInTruck   | cnt |
        countofbdys <- 0.
        qcnt <- '       '.
        qname <- '      '.
        self pr:[ ('  [ ',      (self printString),
                                "adds the time."
         ' ]    ---------------------------------' )].

        iTask   reportOnWorkers.

        cnt <-          (self countTruck: inTruck).

        (' @    unloadW=', (unloadW printString) )print.

        cnt <- cnt +    (self countWrks: unloadW).
        self append: 'inTrk' to: cnt.

        (' @    helpers=', (helpers printString) )print.

| listOutTruck | tl que wkr tsk b cnt |

        cnt <-          (self countQue: loadQ).

        (' @    loadW=', (loadW printString) )print.

        cnt <- cnt +    (self countWrks: loadW).
        self  append: 'load' to: cnt.

        cnt <-          (self countTruck: outTruck).
        cnt <- cnt +    (self countTruckQ: exitQ).
```

```
        self append: 'outTrk' to: cnt.

        self append: 'Bdys' to: (self countIs).
        self append: 'ThruPut' to: (goneQ size).

        self countIs > maxBodys
                ifTrue:  [ fowarding <- true  ]
                ifFalse: [ fowarding <- false ].
        self qcntReport.

| listOutTC | tl que wkr tsk b cnt |

        self countIs > maxBodys
                ifTrue:  [ fowarding <- true  ]
                ifFalse: [ fowarding <- false ].
        self qcntReport.


]
```

Class DemonTask

```
    "----------------Process to displAy queues. "
Class DemonTask :CollectionClass
        | rate go name deltaBodys nBodys |
[ startUp
        deltaBodys <-   Clock new.
        nBodys     <-   0.
        rate <- 60.
        go <- true.
| prefix: echelon
        ^echelon, 'Demon'.
| rate: r
        rate <- r.
| body: b delta: ck
        deltaBodys <-   deltaBodys + ck.
        nBodys     <-   nBodys     + 1.

| stop
        self listWorkerTask.
        super terminate.

| next
        self procDemon.
        self listWorkerTask.
        [ ' @ Bodys:    ',
                (self timeIs    printString), ' ',
                (deltaBodys     printString), ' ',
                (nBodys         printString), ' ###',
                (self cp        printString) ]display.
        self lightsOut  ifTrue: [
                self sch: (self morning: (self timeIs))
        ]ifFalse:[
                self schedule: rate
        ].
]
```

Class Environment

```
     Class Environment :Identity      | rootCp |
[ startUp       "CI 1"
        | res | "store the restart instance."
        ' ' print.

        self sch: ((res <- (self
                        create: ReStart)) start).

        rootCp <- res go:self.

| prefix
        ^'E'
| next            |wl w|
        [ '| reStart    timeIs=',
                (self timeIs printString) ]display.

        (wl <- (self create: ReStart)
                reStart: self) notNil ifTrue: [
                [ 'workLoad = ',
                (wl printString) ] display.
                rootCp reStart: wl.
                ^true
        ].
        self terminateNow.
]
```

Class GenTruck

```
    "       Adds bodies to truckQueue."
Class GenTruck          :CollectionClass
        | queue rate bdys fill uniFill
                trucks timeTo beenToCP |
[ startUp
        beenToCP <- 0.
        rate <- 25.     "default self schedule rate"
        bdys <- 0.      "number of bodies to consume."
        trucks <- 0.    "number of self schedules."
        timeTo <- 10.   "how long get to CP."
        fill <- 0.
| timeTo: t
        timeTo <- t.

| setIP
        beenToCP <- 2.
| setCP
        beenToCP <- 1.

| rate: r
        rate <- r.
| rate
        ^rate
| prefix: echelon
        ^echelon, 'gen'

"       deliver all bodies in less than 6 hours."
| bdys: b
        bdys <- b.                          "work load."
        (bdys < 7200 and: [ bdys > 0] ) ifTrue: [
                rate <- 7200 / bdys roundTo: 1.
        ]ifFalse:[
                rate <- 1.
        ].
        [ 'Nbodys = ', (bdys printString) ]display.
| trucks: b
        trucks <- b.
| queue:   q
        queue <- q.

| fill: min to: max     | a b |
        b <- max + 0.5.
        fill <- min.
        uniFill <- Uniform new initialize
                        from: a to: b.

"Scheduling"
| next
        ( bdys > 0 ) ifTrue: [
```

```
            beenToCP > 0 ifTrue: [
                bdys > 24 ifTrue:
                        [bdys <- bdys - 24.
                        fill <- 24.
                        self schedule: rate]
                        ifFalse: [fill <- bdys  ].
                self makeTruck.
            ] ifFalse: [
                fill <- uniFill next roundTo:1.
                ( (bdys - fill) >= 0 )
                        ifTrue:  [
                        bdys <- (bdys - fill).
                        self makeTruck.
                        self schedule: rate
                ] ifFalse: [
                        fill <- bdys.
                        self makeTruck.
                        self print:[ ( (self printString),
                                ' Terminates.' )].
            ]   ]
        ] ifFalse: [
                ( (trucks <- trucks - 1) < 0 ) ifTrue: [
                        self print:[ ( (self printString),
                                ' Terminates.' )].
                ]ifFalse: [
                        self makeTruck.
                        self schedule: rate
        ]       ].

"Private"
| makeTruck     | t2 |
        self nightTime ifTrue: [
                t2 <- timeTo * 2.
        ]ifFalse:[
                t2 <- timeTo.
        ].
 ( '>> makeTruck        t2=',   (t2  printString),
        '       fill=', (fill  printString) )print.

        ^(((((self create: Truck)  setCP: (self cp)
                ) goingTo: queue
             ) beenToCP: beenToCP
           ) fill: fill
         ) schedule: t2.  "How long it takes to
                           get to the queue."
]
```

Class GlobalData

```
     Class GlobalData        :UserData
    |   evQ          "The environment Queue."
        uniqueName   "Dictionary of unique name counters."
        curTime      "Current Time Clock."
        prTime       "Clock as of the last print occured."
    |
[ new
        evQ <- SList new.
        uniqueName <- Dictionary new.
        curTime <- Clock new incHour: 10.
        prTime <- Clock new.

| print: s
        ( (prTime = curTime) not) ifTrue: [
                ( ' @ ', (curTime printString) ) print.
                prTime set: curTime.
        ].
        ( ' @    ', s )print.

"Accesing"
| timeIs
        ^curTime
| eventQ
        ^evQ
| userData
        ^self.

"Setting"
| timeIs: t
        ^curTime <- t.
"----   Return a post fix to make the name unique."
| genNum: prefix         | n |
        ( n <- uniqueName at: prefix
             ifAbsent: [nil]  ) isNil ifTrue: [ n <- 0 ].
        uniqueName at: prefix put: (n <- n + 1).
        ^ n.
]
```

Class Identity

```
     Class   Identity          :UserAccess
        | name globalData |
[ idStartUp: gd
        "Every one must get access to the data store."
        globalData <- gd.
        self userSetUp: (globalData userData).
        name <- self class printString.
| idStartUp: gd name: aName
        "Every one must get access to the data store."
        globalData <- gd.
        self userSetUp: (globalData userData).
        name <- (aName, ((globalData genNum: aName)
                                printString) ).

"    make a new instance if a simulation object with the
        name fixed up and the global data in place.     "
| create: aClass          | inst pre |

        inst <- aClass new.
        ( (pre <- inst prefix: (self printString)) isNil )
            ifTrue: [ pre <- inst prefix ].

        inst    idStartUp: (self globalData)
                name:      pre.

        inst startUp.
        ^inst

"report output"
| pr: block
        ^ self
| print: block
        globalData print: (block value).

"Accessing"
| globalData
        ^globalData
| timeIs
        ^globalData timeIs

| timeIs: t
        ^globalData timeIs: t

| asString
        ^name

| printString
        ^name

| name: n
```

```
        name<-n.

| prefix
        ^self class printString
| prefix: mother
        ^nil


"Scheduling"
| schedule
        (globalData eventQ)      add: (globalData timeIs)
                                 put: self

| sch: clock
        (globalData eventQ)      add: clock
                                 put: self
| scheduleAfter: clock
        self sch: (clock + (globalData timeIs)).

| scheduleNow
        self sch: (globalData timeIs)

| schedule: hour after: min      | c |
        ( min < 100 and: [ hour < 24 ] )
            ifTrue: [ self sch:(( ( (Clock new
                                  )set: (globalData timeIs)
                                )incMin: min
                              )incHour: hour)
        ]  ifFalse: [ 'schedule error' print ].
| schedule: min
        self sch: ( ( (Clock new
                    )set: (globalData timeIs)
                  )incMin: min)


| schSec: sec
        self sch: ( ( (Clock new
                    )set: (globalData timeIs)
                  )incSec: sec)
]
```

Class IdleTask

```
     Class IdleTask  :Identity
"Data"|
        cp
        idlers
        timesIdle maxIdle go    "for ending"
        allWorkers
      |
[ prefix: echelon
        ^ echelon, 'Idle'
| startUp
        idlers          <- List new.
        allWorkers      <- List new.
        maxIdle <- 12.
        go <- true.
        timesIdle <- 0.
        self start.
| setCP: cpoint
        cp <- cpoint
| personal: n
         [ 'personal = ', (n printString) ]display.
                n timesRepeat: [ allWorkers
                add:(idlers add: ( (cp create: Worker)
                                        setCP:cp )) ].

| reportOnWorkers |s n|
        s <- ''.
        n <- 1.
        allWorkers do: [ :w |
            s <- s, '   ', (w printTime).
            (n <- n + 1) > 3 ifTrue: [
                (' @', s )print.
                s <- ''.
                n <- 1.
        ]   ].
        n > 1 ifTrue: [ (' @', s )print ].


| idleWorker: w
        w sleeping ifFalse: [
          idlers isEmpty ifTrue: [
                 maxIdle > 0 ifTrue: [
                   go ifTrue:  [ self schedule:  5 ]
                        ifFalse: [ self schedule: 60.
                                   go <- true.
                ]   ]   ].
                w setIdle.
                idlers add: w.
        ].
```

```
| idlers
        ^idlers.

| next          | rt |
        timesIdle <- timesIdle + 1.
        rt <- cp reTask: idlers.

        rt ifTrue: [ timesIdle <- 0. ].
        timesIdle >= maxIdle ifTrue: [
            self stopable ifTrue: [
                maxIdle <- 0.
                cp stop.
                [ (self printString),
                        ' terminates.' ]display.
            ].
            go <- false.
        ].
]
```

Class IntermediatePoint

```
     Class IntermediatePoint :CpTrucks
"Data"|
        evacW   ckIdW   idW     "Workers doing tasks"
        packW   moveW   ddW     miscW
        "Places to put bodys."
        evacQ   ckIdQ   idQ     goneQ
        packQ   moveQ   ddQ     miscQ loadQ
        taskList        "a list of triples
                        (see makeTaskList)."
        cpList          "a list if objects
                        for restarting."
        battleZone      "Lights out at night."
"Methods"|
[ prefix: echelon       | ch |
 ['     IP | echelon = ', echelon ]display.

        ch <- echelon at:1.
        ch == $x ifTrue: [
        ^(echelon copyFrom: 2 to: (echelon size)), 'i'
        ].
        ^'xxx'

| startUp
 ['     IP | startUp    ', (self printString) ]display.

        super startTrucks.
        self startQueues.

        self taskList: (taskList _ self makeTaskList).
        battleZone <- true.

| truckFull: bdys at: clock
        (((((((self create: GenTruck) setCP: self
                  )queue: (self enterQ)
                ) setCP          "beenToCp"
              ) timeTo: 180      "180 minutes"
            ) bdys: bdys
          ) rate: 0
        ) sch: clock.
( '^ truckFull: ', (bdys printString), ' at: ',
                        (clock printString) )print.
        ^self.

| reStart: wl   | bdys trk cl |
        [ 'IP | cpList = ', (cpList printString)
                                ]display.
        [ 'IP | wl = ', (wl printString) ]display.
        cl <- List new.
        cpList do: [ :c | cl add: c ].  "Copy."
        wl do: [ :x |
```

```
            [ 'IP|      x = ', (x printString),
                '       cl = ', (cl printString)
                                ]display.
            cl first = self ifTrue: [
                        (cl remove) selfReStart: x
                    ]ifFalse:[
                x first = 'I' ifTrue: [ cpList add:     (
                        ( (self create: IntermediatePoint
                          )     truckNext: (self enterQ)
                        )       deceased: x             )
                    ]ifFalse:[
                        x first = 'C' ifTrue: [ cpList add: (
                        ( (self create: CollectionPoint
                          )     truckNext: (self enterQ)
                        )       deceased: x             )
                        ]ifFalse:[
                        (cl remove) reStart: x
        ]   ]   ]   ].


| selfReStart: w        | trk bdys ch |
        [ 'IP|  w = ', (w printString) ]display.
        ch <-   w  first.
        bdys <- w  next.

        trk <-  (((((self create: GenTruck) setCP: self
                      )queue: (self enterQ)
                    ) fill: (self truckMin)
                      to: (self truckFull)
                  ) bdys: bdys
                ) schedule: 1.

        ( ( ( ( (self create: GenTruck) setCP: self
              )queue: (self exitQ)
            ) rate: (trk rate)
          ) trucks: ( bdys / 6 roundTo:1 )
        ) schedule: 30.

| deceased: list reStart: rs    | car |

        list remove.
        [ 'IP list=', (list printString) ]display.
        cpList <- List new.

        car <- list remove.
        [ 'IP car=', (car printString) ]display.
        car first = 'I' ifTrue: [
                cpList add:  (self selfStart: car).
        ]ifFalse:[
                [ 'error: IP must be first.' ]display.
        ].
        [ list first notNil ] whileTrue: [
            car <- list remove.
```

```
            [ 'IP car=', (car printString) ]display.
            car first = 'I' ifTrue: [
                cpList add: (
                        ( (self create: IntermediatePoint
                          )     truckNext: (self enterQ)
                        )       deceased: car   )
            ]ifFalse:[
                car first = 'C' ifTrue: [
                    cpList add: (
                        ( (self create: CollectionPoint
                            )   truckNext: (self enterQ)
                          )     deceased: car   )
                ]ifFalse:[
                    ['IP error: bad input car= ',
                                (car printString),
                        '       cdr= ', (list printString)
                                        ]display.

        ]   ]   ].
        ['cpList = ', (cpList printString)]display.
        rs fowardTrucksTo: self.

| selfStart: list       | foo pers bdys time trk|

        foo <- list remove.
        pers <- list remove.
        bdys <- list remove.
        time <- list remove.

        trk <-  (((((self create: GenTruck) setCP: self
                      )queue: (self enterQ)
                    ) fill: (self truckMin)
                      to: (self truckFull)
                  ) bdys: bdys
                ) schedule: 1.

        ( ( ( ( ( (self create: GenTruck) setCP: self
                ) queue: (self exitQ)
              ) timeTo: time
            ) rate: (trk rate)
          ) trucks: 20
        ) schedule: 30.

        ( ( (self iTask: (self create: IdleTask))
                                setCP: self
          ) personal: pers
        ) scheduleNow.

        ( (self demon: (self create: DemonTask))
                                setCP: self
        ) schedule: 10.

| makeTaskList  | l |
        l <- List new.
```

```
        l add: ( self makeTL: evacQ
                        with: evacW     task: Evac).
        l add: ( self makeTL: ckIdQ
                        with: ckIdW     task: CkId).
        l add: ( self makeTL: moveQ
                        with: moveW     task: Move).

        l add: ( self makeTL: idQ
                        with: idW       task: Id).
        l add: ( self makeTL: packQ
                        with: packW     task: Pack).

        l add: ( self makeTL: ddQ
                        with: ddW       task: Dd).
        l add: ( self makeTL: miscQ
                        with: miscW     taskN: Misc).
        ^l

        "Make the queues for the Collection Point."
| startQueues
        evacQ   <- self unloadQ.
        idQ     <- List new.
        ckIdQ   <- List new.

        moveQ   <- List new.
        packQ   <- List new.
        ddQ     <- List new.
        miscQ   <- List new.
        goneQ   <- List new.
        loadQ   <- self loadQ.
                                "Workers"
        evacW   <- List new.
        idW     <- List new.
        ckIdW   <- List new.

        moveW   <- List new.
        packW   <- List new.
        ddW     <- List new.
        miscW   <- List new.

| lightsOut
        battleZone ifTrue: [    ^self nightTime ].
        ^false.

"----------------
#     NEXT
"
| procEvac: tObj | wkr bdy |
        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        bdy notNil ifTrue:[
```

```
            bdy beenToCP ifTrue:[
                self    put: bdy
                        by: wkr
                        on:  ckIdQ
                        msg: 'completes Evac#, on'.
            ]ifFalse:[
                self    put: bdy
                        by: wkr
                        on:  idQ
                msg: 'completes Evac form DD1077, on'.
        ]   ].

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy: wkr from: evacQ
                        msg: 'assigns Evac # to'.
        ].
        self free: tObj when: bdy from: evacW.

        ^bdy.



| procId: tObj          | wkr bdy |
        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy
                by: wkr
                on: packQ
                msg: 'completes Id and DD forms, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.                 "reTask"
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   idQ
                        msg:    'starts to Id'.
        ].
        self free: tObj when: bdy from: idW.
        ^bdy.



| procCkId: tObj                | wkr bdy |
        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy
                by: wkr
```

```
                on: moveQ
                msg: 'completes CkId and DD forms, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.                         "reTask"
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   ckIdQ
                        msg:    'starts to CkId'.
        ].
        self free: tObj when: bdy from: ckIdW.
        ^bdy.


| procPack: tObj        | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  ddQ
                msg: 'completes Moving, of '.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.                         "reTask"
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   packQ
                        msg: 'start to Pack and Move'.
        ].
        self free: tObj when: bdy from: packW.
        ^bdy.

| procMove: tObj        | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  ddQ
                msg: 'finishes Moving, of '.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   moveQ
                        msg:    'start to Move'.
        ].
        self free: tObj when: bdy from: moveW.

        ^bdy.

| procDd: tObj  | wkr bdy |
```

```
        wkr <- tObj workers first.
        (bdy <- tObj body) notNil ifTrue: [ bdy setIP ].
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  loadQ
                msg:    'completes DD175, for'.

        self    put: bdy   by: wkr  on: miscQ.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   ddQ
                msg:    'starts to fill out DD175 for'.
        ].
        self free: tObj when: bdy from: ddW.
        ^bdy.

| procMisc: tObj        | wkr bdy |

        wkr <- tObj workers first.
        tObj body: nil.

        self    put: bdy   by: wkr  on: goneQ.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   miscQ
                msg:    'does the miscellaneous task.'.
        ].
        self free: tObj when: bdy from: miscW.
        ^bdy.

"----------------"
| list
        self listInTruck.
        self listTask: taskList.
        (' @    goneQ   size=',
                (goneQ size printString) )print.
        self listOutTruck.

| listWorkerTask
        self listInTruck.
        self listTask: taskList.
        (' @    goneQ   size=',
                (goneQ size printString) )print.
        self listOutTruck.
        (' @    load=',
                (self countIs printString) )print.
        self reTask.
]
```

```
Class Obj


     "This is the root of all simulation objects"
"Note that objects can be listed with 'Obj list'."
Class Obj

[ new
        ^self.
]
```

Class Probability

```
     Class   Probability     :Obj
        | randnum |
[ initialize
        randnum <- Random new randomize.
"       randnum <- Random new.  "
| next
        ^self sample: randnum next.
| first
        ^-1.
| getRandom
        ^ randnum next.
]
```

Class ReStart

```
     Class ReStart :SuperReStart
        | envObj wrkLoad |
[ startUp
        ^self.

"---------------- Work Load at time=0."
| cp0
        ^(self cpWrks: 8 bdys: 3 routeTime: 7).
| cp1
        ^(self cpWrks: 8 bdys: 87 routeTime: 7).

"---- TC "
| dIp
        ^self   1: 'I'
                1: (self ipWrks: 6 bdys: 4 routeTime: 180)
                1: (self cpWrks: 4 bdys: 7 routeTime: 180)

| cIp   | 1 |

        ^self   1: 'I'
                1: (self ipWrks: 6 bdys: 4 routeTime: 180)
                1: (self dIp )



|ipCI2
        ^self   1: (self ipBdys:  6 )   "i1"
                1: (self cpBdys: 50 )   "c1"
                1: (self cpBdys: 50 )   "c2"
                1: (self cpBdys: 46 )   "arc [i2c1] -> C3"
|ipCI3
        ^self   1: (self ipBdys:  6 )
                1: (self cpBdys: 60 )
                1: (self cpBdys: 28 )
                1: (self cpBdys: 59 )   "arc"
|ipCI4
        ^self   1: (self ipBdys:  6 )
                1: (self cpBdys: 56 )
                1: (self cpBdys: 28 )
                1: (self cpBdys: 31 )   "arc"
|ipCI5
        ^self   1: (self ipBdys:  6 )
                1: (self cpBdys:  0 )
                1: (self cpBdys: 27 )
                1: (self cpBdys: 45 )   "arc"
                1: (self cpWrks: 8
                          bdys: 14
                     routeTime: 180)    "[c3] -> C4"

|ipCI6
        ^self   1: (self ipBdys:  6 )
```

```
                l: (self cpBdys:  0 )
                l: (self cpBdys: 39 )
                l: (self cpBdys: 16 )   "arc"
                l: (self cpBdys: 39 )   "C4"
|ipCI7
        ^self   l: (self ipBdys:  6 )
                l: (self cpBdys:  0 )
                l: (self cpBdys: 17 )
                l: (self cpBdys: 56 )   "arc"
                l: (self cpBdys: 17 )   "C3"
|ipCI8
        ^self   l: (self ipBdys:  6 )
                l: (self cpBdys: 11 )
                l: (self cpBdys: 20 )
                l: (self cpBdys: 36 )   "arc"
                l: (self cpBdys:  0 )   "C3"
|ipCI9
        ^self   l: (self ipBdys:  6 )
                l: (self cpBdys: 25 )
                l: (self cpBdys:  8 )
                l: (self cpBdys:  0 )   "arc"
                l: (self cpBdys: 46 )   "C3"
|ipCI10
        ^self   l: (self ipBdys:  6 )
                l: (self cpBdys: 18 )
                l: (self cpBdys: 19 )
                l: (self cpBdys:  7 )   "arc"
                l: (self cpBdys: 35 )   "C3"

"---- TC"
| ci2TC
        ^self   l: (self tcBdys: 4 )
                l: (self cpBdys: 5 )
                l: (self
                        l: (self ipBdys: 3)
                        l: (self
                                l: (self ipBdys: 1)
                                l: (self cpBdys: 2)
                )       )
"---------------- Access."

| start
        self timeIs: (self ciltime).
        self print: [ '--------   CI 1   ---------' ].
        ^self ci2time.

| reStart: env
        envObj <- env.
        [(self printString), ' ', (envObj printString)
                                        ] display.
        self testCI2    ifTrue: [^wrkLoad].
        self testCI3    ifTrue: [^wrkLoad].
        self testCI4    ifTrue: [^wrkLoad].
```

```
        self testCI5    ifTrue: [^wrkLoad].
        self testCI6    ifTrue: [^wrkLoad].
        self testCI7    ifTrue: [^wrkLoad].
        self testCI8    ifTrue: [^wrkLoad].
        self testCI9    ifTrue: [^wrkLoad].
        self testCI10   ifTrue: [^wrkLoad].

        self print: [ '-----  Terminate.  ------' ].
        ^nil

"---------------- Private."

| testCI2                | nextTime |
        nextTime<- self ci3time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 2 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci2.            "more Workload."
        ^true                           "ok, Run"

| testCI3                | nextTime |
        nextTime<- self ci4time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 3 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci3.            "more Workload."
        ^true                           "ok, Run"

| testCI4                | nextTime |
        nextTime<- self ci5time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 4 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci4.            "more Workload."
        ^true                           "ok, Run"

| testCI5                | nextTime |
        nextTime<- self ci6time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 5 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci5.            "more Workload."
        ^true                           "ok, Run"

| testCI6                | nextTime |
        nextTime<- self ci7time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 6 -----' ].
```

```
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci6.            "more Workload."
        ^true                           "ok, Run"

| testCI7               | nextTime |
        nextTime <- self ci8time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 7 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci7.            "more Workload."
        ^true                           "ok, Run"

| testCI8               | nextTime |
        nextTime<- self ci9time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 8 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci8.            "more Workload."
        ^true                           "ok, Run"

| testCI9               | nextTime |
        nextTime<- self ci10time.
        (nextTime <= self timeIs) ifTrue:
                [^false ].      "been here before."
        self print: [ '----- CI 9 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci9.            "more Workload."
        ^true                           "ok, Run"

| testCI10              | nextTime |
        nextTime<- self stopTime.
        (nextTime <= self timeIs) ifTrue:
                [ ^false ].     "been here before."
        self print: [ '----- CI 10 -----' ].
        envObj sch: nextTime.           "Store Time."
        wrkLoad <- self ci10.           "more Workload."
        ^true                           "ok, Run"
]
```

Class RestartTools

```
     Class RestartTools      :Identity
[ tcWrks: wrks  bdys: bdys
        "---------------- List makers."
        ^ self l: 'T' l: wrks l: bdys
| ipWrks: wrks  bdys: bdys  routeTime: t
        ^ self l: 'I' l: wrks l: bdys l: t
| cpWrks: wrks  bdys: bdys  routeTime: t
        ^ self l: 'C' l: wrks l: bdys l: t
| ipWrks: wrks  bdys: bdys
        ^ self l: 'I' l: wrks l: bdys l: 180
| cpWrks: wrks  bdys: bdys
        ^ self l: 'C' l: wrks l: bdys l: 180

| cpBdys: b
        ^self l: 'p' l: b
| ipBdys: b
        ^self l: (self l: 'i' l: b)

| tcBdys: b
        ^self l: (self l: 't' l: b)

| l: a                      | l |
        l <- List new.
        l add: a.
        ^l
| l: a l: b                 | l |
        l <- List new.
        l add: a.
        l add: b.
        ^l
| l: a l: b l: c            | l |
        l <- List new.
        l add: a.
        l add: b.
        l add: c.
        ^l
| l: a l: b l: c l: d       | l |
        l <- List new.
        l add: a.
        l add: b.
        l add: c.
        l add: d.
        ^l
| l:a l:b l:c l:d l:e       | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
```

```
        ^lst
| l:a l:b l:c l:d l:e l:f        | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
        lst add: f.
        ^lst
| l:a l:b l:c l:d l:e l:f l:g    | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
        lst add: f.
        lst add: g.
        ^lst
| l:a l:b l:c l:d l:e l:f l:g l:h        | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
        lst add: f.
        lst add: g.
        lst add: h.
        ^lst
| l:a l:b l:c l:d l:e l:f l:g l:h l:i    | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
        lst add: f.
        lst add: g.
        lst add: h.
        lst add: i.
        ^lst
| l:a l:b l:c l:d l:e l:f l:g l:h l:i l:j        | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
        lst add: f.
        lst add: g.
        lst add: h.
```

```
        lst add: i.
        lst add: j.
        ^lst
| l:a l:b l:c l:d l:e l:f l:g l:h l:i l:j l:k   | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
        lst add: f.
        lst add: g.
        lst add: h.
        lst add: i.
        lst add: j.
        lst add: k.
        ^lst
| l:a l:b l:c l:d l:e l:f l:g l:h
                        l:i l:j l:k l:l | lst |
        lst <- List new.
        lst add: a.
        lst add: b.
        lst add: c.
        lst add: d.
        lst add: e.
        lst add: f.
        lst add: g.
        lst add: h.
        lst add: i.
        lst add: j.
        lst add: k.
        lst add: l.
        ^lst
]
```

Class Simulation

```
     Class Simulation          :Obj
        | globalData evQ go envObj |
[ startUp
        globalData _ GlobalData new.
        envObj <- ((Environment new
                idStartUp: globalData) startUp).
        evQ _ (globalData eventQ).
        go _ true.

| process        | time process |
        time _ evQ key.
        globalData timeIs: time.
        process _ evQ remove.
        (go and: [ process notNil ]) ifTrue:
        [       go _ globalData nextPrint: process.
                process next.
                ^true
        ].
        [ 'exit go= ', (go printString) ]display.
        smalltalk sh: 'mv _go _stop'.
        ^false
]
```

Class SuperReStart

```
     Class SuperReStart :RestartTools
        | node |
[ go:env

        ^(env create: CollectionPoint)
                                deceased: (self cp)

"
*       ^(env create: IntermediatePoint)
*                               deceased: (self ip)
*                               reStart: self.
*
*       ^(env create: TemporaryCemetery)
*                               deceased: (self tc)
*
*       ^(env create: CollectionPoint)
*                               deceased: (self cp0)
*
*       ^(env create: CollectionPoint)
*                               deceased: (self cp1)
*
*       ^(env create: IntermediatePoint)
*                               deceased: (self dIp)
*       ^(env create: IntermediatePoint)
*                               deceased: (self cIp)
*
"

| cp
        ^self   cpWrks: 8 bdys: 54 routeTime: 7.

| tc
        ^self   1: (self tcWrks: 6 bdys:  6 )

| ip
        ^self   1: 'I'
                1: (self ipWrks: 7 bdys:  0
                                routeTime: 180)  "il"

| fowardTrucksTo: p
        node <- p.
        self trk1.
        self trk2.



"
^  messages to place CP trucks on the event queue
^  have the form:
^
^  node truckFull: nBdys at:(((Clock new)day: dd)hour: hh)
```

```
^
"
| trk1
        node truckFull: 31 at: (((Clock new)day:4)hour:11).
        node truckFull: 5 at: (((Clock new)day:4)hour:12).
        node truckFull: 27 at: (((Clock new)day:5)hour:09).

        node truckFull: 14 at: (((Clock new)day:5)hour:11).
        node truckFull: 14 at: (((Clock new)day:6)hour:09).
        node truckFull: 1 at: (((Clock new)day:6)hour:11).
| trk2
        node truckFull: 30 at: (((Clock new)day:6)hour:12).


"----------------       Other Work Loads.       "
| ci2
        ^self cpBdys: 20
| ci3
        ^self cpBdys: 24
| ci4
        ^self cpBdys: 17
| ci5
        ^self cpBdys: 45
| ci6
        ^self cpBdys: 25
| ci7
        ^self cpBdys: 48
| ci8
        ^self cpBdys: 66
| ci9
        ^self cpBdys: 104
| ci10
        ^self cpBdys: 0


"----------------       Times of each CI."
| ciltime                          "StartUp Time"
        ^((Clock new day:0)  hour:04)  min:0

| ci2time
        ^((Clock new day:0)  hour:22)  min:0

| ci3time
        ^((Clock new day:1)  hour:04)  min:0

| ci4time
        ^((Clock new day:1)  hour:16)  min:0
| ci5time
        ^((Clock new day:2)  hour:04)  min:0
| ci6time
        ^((Clock new day:2)  hour:16)  min:0
| ci7time
        ^((Clock new day:3)  hour:04)  min:0
```

```
| ci8time
        ^((Clock new day:3)  hour:16)  min:0
| ci9time
        ^((Clock new day:4)  hour:04)  min:0
| ci10time
        ^((Clock new day:4)  hour:16)  min:0
| stopTime                          "terminate Time"
        ^((Clock new day:10) hour:00)  min:0

]
```

Class Task

```
     Class Task      :CollectionClass
     |  body
        workerList      "A List of the
                        workers moving the body."
        name
        uniTime
     |
[ startUp
        body <- nil.
        uniTime <- Uniform new var20: (self taskTime).

| taskClock     | tck |
        tck <- uniTime taskClock.
        ^tck

| name: n
        ^name <- n
| printString
        ^name.
        body isNil ifTrue:  [ ^name ]
                   ifFalse:[ ^( name,
                        '(', (body printString), ')' )].
| body: b
        body <- b
| body
        ^body
"----------------"
| setWorking    | w |
        w <- workerList first.
        [ w notNil ] whileTrue:
            [   w setWorking.
                w <- workerList next
            ].
| workers
        ^workerList.
| workers: wl
        workerList <- wl.

| reSchedule: wObj      | t |
        body notNil ifTrue: [
                (t <- wObj taskClock) notNil
                    ifTrue:[ wObj scheduleAfter: t ]
                    ifFalse:
                        [ 'error...taskClock = nil' print.
                          wObj schedule: (wObj taskTime) ].
        ] ifFalse: [
                self reTask: workerList.
        ]
]
```

Class TemporaryCemetery

```
     Class TemporaryCemetery :CpTrucks
"Data"|
        evacW   ckIdW   idW     "Workers doing tasks"
        packW   moveW           "Workers doing tasks"
        disRobeW        fingerpW
        dTailIdW        shroudW
        platesW         digW
        mv2siteW        dd3x5W
        shipPeW         inCoverW

        evacQ   ckIdQ   idQ     "Places to put bodys."
        packQ   moveQ           "Places to put bodys."
        disRobeQ        fingerpQ
        dTailIdQ        shroudQ
        platesQ notDugQ digQ  dugQ
        mv2siteQ        dd3x5Q
        shipPeQ         inCoverQ
        groundQ

        taskList  "a list of triples (see makeTaskList)."
        cpList    "a list if objects for restarting."

"Methods"|
[ prefix
        ^'t'
| startUp
 ['TC | startUp ', (self printString) ]display.

        super startTrucks.
        self startQueues.

        0 timesRepeat: [ self addBodyOn: evacQ ].
        0 timesRepeat: [ self addBodyOn: mv2siteQ ].

        self taskList: (taskList <- self makeTaskList).

| truckFull: bdys at: clock
        ( ( ( ( ( ( (self create: GenTruck) setCP: self
                  )queue: (self enterQ)
                ) setCP                    "beenToCp"
              ) timeTo: 180                "180 minutes"
            ) bdys: bdys
          ) rate: 0
        ) sch: clock.
( '^ truckFull: ', (bdys printString),
        ' at: ', (clock printString) )print.
        ^self.

| reStart: wl   | bdys trk w |
        [ 'TC | cpList = ', (cpList printString)
```

```
                                        ]display.

        cpList do: [ :x |
                [ 'TC | wl = ', (wl printString)
                                        ]display.
                w <- wl remove.
                [ 'TC|  x = ', (x printString),
                  '     w = ', (w printString)
                                        ]display.
                x = self ifTrue:
                        [ self selfReStart: w ]
                         ifFalse: [ x reStart: w ].
        ].

| selfReStart: w          | trk c b |
        [ 'TC|  w = ', (w printString) ]display.

        c <- w first.
        b <- w next.


        ( ( ( ( (self create: GenTruck) setCP: self
              ) queue: (self enterQ)
            ) fill: (self truckMin)
              to: (self truckFull)
          ) bdys: b
        ) schedule: 1.


| deceased: list reStart: rs     | car |
        [ 'TC list=', (list printString) ]display.
        cpList <- List new.
                                "do self first"
        car <- list remove.
        [ 'TC car=', (car printString) ]display.
        car first = 'T' ifTrue: [
                cpList add:     (self selfStart: car).
        ]ifFalse:[
                [ 'error: TC must be first.' ]display.
        ].
        [ list first notNil ] whileTrue: [
            car <- list remove.
            car first = 'I' ifTrue: [
                cpList add: (( (self
                        create: IntermediatePoint
                          )     truckNext: (self enterQ)
                        )       deceased: car   )
            ]ifFalse:[
                car first = 'C' ifTrue: [
                   cpList add:  ( ( (self
                        create: CollectionPoint
                            )   truckNext: (self enterQ)
                          )     deceased: car   )
```

```
                ]ifFalse:[
                    ['TC error: bad input car= ',
                                (car printString),
                          '     cdr= ', (list printString)
                                        ]display.
        ]   ]   ].
        ['cpList = ', (cpList printString)]display.
        rs fowardTrucksTo: self.

| selfStart: list       | foo pers bdys |
        foo <- list remove.
        pers <- list remove.
        bdys <- list remove.

        ( ( ( ( (self create: GenTruck) setCP: self
              ) queue: (self enterQ)
            ) fill: (self truckMin) to: (self truckFull)
          ) bdys: bdys
        ) schedule: 1.

        ( ( (self iTask: (self create: IdleTask))
                                setCP: self
          ) personal: pers
        ) scheduleNow.

        ( (self demon: (self create: DemonTask))
                                setCP: self
        ) schedule: 10.

| makeTaskList  | l |
        l <- List new.

        l add: ( self makeTL: evacQ
                        with: evacW
                        task: Evac      ).
        l add: ( self makeTL: digQ
                        with: digW
                        taskN: Dig      ).

        l add: ( self makeTL: ckIdQ
                        with: ckIdW
                        task: CkId      ).
        l add: ( self makeTL: moveQ
                        with: moveW
                        task: Move      ).

        l add: ( self makeTL: idQ
                        with: idW
                        task: Id        ).
        l add: ( self makeTL: packQ
                        with: packW
                        task: Pack      ).
```

```
        l add: ( self makeTL: disRobeQ
                        with: disRobeW
                        task: DisRobe   ).
        l add: ( self makeTL: fingerpQ
                        with: fingerpW
                        task: Fingerp   ).
        l add: ( self makeTL: dTailIdQ
                        with: dTailIdW
                        task: DTailId   ).
        l add: ( self makeTL: shroudQ
                        with: shroudW
                        task: Shroud    ).
        l add: ( self makeTL: platesQ
                        with: platesW
                        task: Plates    ).
        l add: ( self makeTL: mv2siteQ
                        with: mv2siteW
                        task: Mv2site   ).
        l add: ( self makeTL: dd3x5Q
                        with: dd3x5W
                        task: Dd3x5     ).
        l add: ( self makeTL: shipPeQ
                        with: shipPeW
                        taskN: ShipPe   ).
        l add: ( self makeTL: inCoverQ
                        with: inCoverW
                        task: InCover   ).

        ^l

        "make the queues for the Collection Point."
| startQueues
        evacQ           <- self unloadQ.
        idQ             <- List new.
        ckIdQ           <- List new.
        moveQ           <- List new.
        packQ           <- List new.
        disRobeQ        <- List new.
        fingerpQ        <- List new.
        dTailIdQ        <- List new.
        shroudQ         <- List new.
        platesQ         <- List new.
        digQ            <- List new.
        dugQ            <- List new.
        notDugQ         <- List new.
        mv2siteQ        <- List new.
        dd3x5Q          <- List new.
        shipPeQ         <- List new.
        inCoverQ        <- List new.
        groundQ <-       List new.
                                "Workers"
        evacW   <- List new.
        idW     <- List new.
```

```
        ckIdW   <- List new.
        moveW   <- List new.
        packW   <- List new.
        disRobeW        <- List new.
        fingerpW        <- List new.
        dTailIdW        <- List new.
        shroudW         <- List new.
        platesW         <- List new.
        digW            <- List new.
        mv2siteW        <- List new.
        dd3x5W          <- List new.
        shipPeW         <- List new.
        inCoverW        <- List new.

| lightsOut                     "No lights out."
        ^false

"----------------
#     NEXT
"
| procEvac: tObj | wkr bdy |

        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        bdy notNil ifTrue:[
            self        put: bdy        by: wkr
                        on:  digQ.
            bdy beenToCP ifTrue:[
                self    put: bdy        by: wkr
                        on:  ckIdQ
                        msg: 'completes Evac#, on'.
            ]ifFalse:[
                self    put: bdy        by: wkr
                        on:  idQ
                msg: 'completes Evac form DD1077, on'.
        ]   ].

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   evacQ
                        msg:    'assigns Evac # to'.
        ].
        self free: tObj when: bdy from: evacW.

        ^bdy.



| procId: tObj          | wkr bdy |
        wkr <- tObj workers first.
```

```
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on: packQ
                msg: 'completes Id and DD forms, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.                 "reTask"
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   idQ
                        msg:    'starts to Id'.
        ].
        self free: tObj when: bdy from: idW.
        ^bdy.


| procCkId: tObj                | wkr bdy |
        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on: moveQ
                msg: 'completes CkId and DD forms, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.                 "reTask"
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   ckIdQ
                        msg:    'starts to CkId'.
        ].
        self free: tObj when: bdy from: ckIdW.
        ^bdy.


| procPack: tObj        | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  disRobeQ
                msg: 'completes Moving, of'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.                 "reTask"
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   packQ
                        msg:    'start to Pack and Move'.
        ].
```

```
        self free: tObj when: bdy from: packW.
        ^bdy.

| procMove: tObj        | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  disRobeQ
                msg: 'finishes Moving, of'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   moveQ
                        msg:    'start to Move'.
        ].
        self free: tObj when: bdy from: moveW.
        ^bdy.

| procDisRobe: tObj     | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  fingerpQ   msg:    'has disrobed'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   disRobeQ
                        msg:    'starts to disrobed'.
        ].
        self free: tObj when: bdy from: disRobeW.
        ^bdy.


| procFingerP: tObj     | wkr bdy |

        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  dTailIdQ
                msg:    'completes FingerP, on'.

        ( self testIdle: tObj ) ifTrue:[
```

```
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   fingerpQ
                        msg:    'starts to FingerP, of'.
        ].
        self free: tObj when: bdy from: fingerpW.
        ^bdy.


| procDTailId: tObj      | wkr bdy |

        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  shroudQ
                msg:    'completes DTailId, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   dTailIdQ
                        msg:    'starts to DTailId, of'.
        ].
        self free: tObj when: bdy from: dTailIdW.
        ^bdy.


| procShroud: tObj      | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  platesQ
                msg:    'completes Shroud, on'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   shroudQ
                        msg:    'starts to Shroud, of'.
        ].
        self free: tObj when: bdy from: shroudW.
        ^bdy.


| procPlates: tObj      | wkr bdy |
```

```
        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  mv2siteQ
                msg:    'completes making plates, on'.
        bdy <- nil.

        ( self testIdle: tObj ) ifFalse: [
            bdy <- self getBy:  wkr     from:   platesQ
                        msg:'starts to make Plates for'.
        ].
        self free: tObj when: bdy from: platesW.
        ^bdy.

| procDig: tObj | wkr bdy |

        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  dugQ
                msg:    'completes the hole, for'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   digQ
                        msg: 'starts to dig a hole , for'.
        ].
        self free: tObj when: bdy from: digW.
        ^bdy.


| procMv2site: tObj     | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  dd3x5Q
                msg:    'completes moving to site, with'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   mv2siteQ.
            bdy notNil ifTrue: [
                (dugQ remove: bdy
                        ifAbsent: [ nil ]) isNil ifTrue:[
```

```
                bdy <- self getHole: bdy.
        ]   ]   ].
        bdy notNil ifTrue: [
           self msg:    'starts to move to site, with '
                wkr: wkr        bdy: bdy.
        ].
        self free: tObj when: bdy from: mv2siteW.
        ^bdy.


| getHole: bdy  | b w |
        (b <- dugQ remove) isNil ifTrue: [
            self put: bdy on: notDugQ.
            [   'body( ', (bdy printString),
                ')s hole is not yet pre-pared.' ]display.
            (digQ contain: b) ifFalse: [
                ( (w <- digW first) isNil or:
                    [ (w body == bdy) not ] ) isTrue: [
                    self put: bdy on: digQ
            ]   ].
            ^nil
        ] ifFalse: [
            self put: b on: digQ.
            [   '(',    (bdy printString),
                        ') is to be placed in (',
                        (b printString),    ')s hole.'
                                        ]display.
            ^bdy
        ].
"All bodies that were not ready for the mv2site
are in the notDugQ.  These bodies are put on the mv2siteQ
when the hole is ready (ie. the body is in the dugQ).   "
| procDemon     | l |
        l <- List new.
        notDugQ do: [ :b |
                l add: b.
                self put: b     on: mv2siteQ.
        ].
        l do: [ :b | notDugQ remove: b ].


| procDd3x5: tObj       | wkr bdy |

        wkr <- tObj workers first.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  inCoverQ
                msg:    'completes Dd3x5, on'.

        self    put: bdy        by: wkr
                on:  shipPeQ.
```

```
        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   dd3x5Q
                        msg:    'starts to Dd3x5, of'.
        ].
        self free: tObj when: bdy from: dd3x5W.

        ^bdy.



| procShipPe: tObj      | wkr bdy |

        wkr <- tObj workers first.
        tObj body: nil.

        self    msg: 'completes PE for shipment for'
                wkr: wkr        bdy: bdy.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   shipPeQ
                        msg:    'starts to shipPe, of'.
        ].
        self free: tObj when: bdy from: shipPeW.

        ^bdy.


| procInCover: tObj     | wkr bdy |

        wkr <- tObj workers.
        bdy <- tObj body.
        tObj body: nil.

        self    put: bdy        by: wkr
                on:  groundQ
                msg:    'completes covering the grave of'.

        ( self testIdle: tObj ) ifTrue:[
            bdy <- nil.
        ]ifFalse: [
            bdy <- self getBy:  wkr     from:   inCoverQ
                        msg: 'starts to put into grave for'.
        ].
        self free: tObj when: bdy from: inCoverW.

        ^bdy.
```

```
"-----------------"
| listQs
  (' @   dugQ    ', (dugQ printString) )print.
  (' @   notDugQ          ', (notDugQ printString) )print.
  (' @   groundQ         size=', (groundQ size printString),
        '                work load=',
                         (self countIs printString) )print.
        ' @     ---------------------------------' print.

| list
        self listInTruck.
        self listTask: taskList.
        self listQs.

| listWorkerTask
        self listInTruck.
        self listTask: taskList.
        self listQs.

        self append: 'Bdys' to: (self countIs).
        self append: 'ThruPut' to: (groundQ size).

        self listOutTC.
        self reTask.
]
```

Class Truck

```
     Class Truck     :CollectionClass
    |   contents         "Where the bodies go."
        destination      "Unloading location."
        beenToCP
    |
[ startUp
        beenToCP <- 0.
        contents <- List new.
| fill: n       | time  bdy |
        (' | fill: ', (n printString),
                '       beenToCP = ',
                (beenToCP printString) )print.
        time <- (Clock new) set: (self timeIs).
        n timesRepeat:
                [ bdy <- ( (self create: Body
                           ) beenToCP: beenToCP
                         ) deathAt: time.
                  contents add: bdy.
                ].
        self print:[ (  (self printString),
                ' drives toward collection station.' )].
        ^self.
| printString
        ^super printString, (contents printString)

| goingTo: loc          "Tell the truck driver where to go."
        destination <- loc.
| prefix: parent     "make parent's name part of the prefix"
        ^parent, 'trk'

| beenToCP: btc
        beenToCP <- btc

"Contents testing/removing/adding"
| remove: aBody
        ^contents removeKey: aBody
| remove
        ^contents remove
| first
        ^contents first
| contents
        ^contents
| isEmpty
        ^contents isEmpty
| size
        ^contents size
| add:  t  put:  bdy
        ^contents add: bdy.
```

```
"Scheduling"
| next
        destination isNil ifTrue: [
                self print: [ ((self printString),
                ' motors off into the sun set.') ]
            ] ifFalse: [
                self print: [ ((self printString),
                                ' arrives.' )].
                destination     add: (self timeIs)
                                put: self.

                destination <- nil.
            ].

| leave: loc timeTo: t2
        self nightTime ifTrue: [
                t2 <- t2 * 2.
        ].
        destination <- loc.
        self schedule: t2.
]
```

Class Uniform

```
     Class   Uniform          :Probability
        | a b |          "uniform distribution on [a,b] "
[ from: start   to: stop         "initialize a and b"

        start < stop
                ifTrue:  [ a <- start.  b <- stop       ]
                ifFalse: [ a <- stop.   b <- start      ].


| var20: min     |n a b s3|
        s3 <- 3 sqrt.
        n <- (min / 5) * s3.
        a <- min - n.
        b <- min + n.

| taskClock      |rnd|
        rnd <- self next.
        ^( (Clock new)
           incMin: (rnd integerPart)
         ) incSec: ((rnd fractionPart * 60) roundTo:1).

| mean
        ^ (a + b) / 2.
| variance
        ^ (b - a) squared  / 12.
| density: x
        (x between: a and: b)
            ifTrue: [^ 1.0 / (b - a) ]
           ifFalse: [^ 0.0].
| sample: x
        ( a = b) ifTrue: [ ^ a]
                 ifFalse: [^ a + (x * (b - a))].
]
```

Class UserAccess

```
     "       This is a superclass of each object.
        It allows access to the user data."
Class UserAccess :Obj
        | userData |
[ userSetUp: ud
        userData _ ud.
                "Give the wake up time."
| morning: ck
        ck _ Clock new: (self timeIs).
        ( ck hour > 8 )ifTrue:[
            ck incDay:  1.
        ].
        ck hour:        8.
        ck min: 0.
        ck sec: 0.
        ^ck

"----------------
Passes on info. about when to terminate the IdleTasks.
"
| setStop
        userData setStop.
| stopable
        ^userData stopable.

                        " add one to counter. "
| foo
        userData foo: ((userData foo) + 1)

| start
        userData start.
| terminateNow
        userData terminateNow
| terminate
        userData terminate.
                        " Describe a result. "
| report
   ( '================ report from ',
                        (self printString)      )print.
   ( 'total number of passengers = ',
                        (userData foo printString) )print.
   '=============================================='print.
]
```

Class UserData

```
     "       This is a storage for global data.
        The data and messages for all global data are
        defined in this class.                          "
Class UserData  :Obj
        | foo stop start n
        stopable        "Set when the last CI is reached."
        sumBodys
        nBodys
        |
[ new
        start <- 0.     "Number of idle tasks running."
        stop <- false.  "Flag the terminate state."
        stopable <- false.
        n <- 0.

        "       This must be defined. May be empty."
        "       nextPrint is run before each event."
| nextPrint: process    |t|
        t <- self timeIs.
        [       '<',            (n      printString),
                '> ',           (t       printString),
                '       ',      (process printString)
                                        ]display.
        n <- n + 1.             "stop conditions"
        stop                    ifTrue: [^false].
        ^true.

"---------------
        Allow the idleTasks to terminate.
"
| setStop
        stopable <- true.
| stopable
        ^stopable.

| start         "count the number of running idle tasks."
        start <- start + 1.
        [ (self printString), ' start = ',
                        (start printString) ] display.

| terminate     "Terminate when all idle tasks stop."
        (start <- start - 1) < 1
                        ifTrue: [ self terminateNow ].
        [ (self printString), ' stop = ',
                        (start printString) ] display.

| terminateNow
        stop <- true
```

```
| foo: f                    " set foo "
       foo <- f
| foo                       " recall foo "
       ^foo
]
```

Class Worker

```
     Class Worker    :CollectionClass
     |
        idleMax idleCount
        taskObj          " what we are doing "
        lastFun lastTime workTime idleTime
        todayWorkTime
        name
        sleeping
     |
[ startUp
        lastTime <- Clock new.
        workTime <- Clock new.
        todayWorkTime <- Clock new.
        idleTime <- Clock new.
        lastFun <- 'null'.
        sleeping <- false.
        name <- super printString.

        taskObj <- nil. " what we are doing "

        idleMax <- 6.    "number of times
                         to wait idle, then quit"
        idleCount <- 0.

| prefix: echelon
        ^(echelon, 'W')
| body
        taskObj notNil ifTrue: [ ^(taskObj body) ].
        ^nil
| printString   |bdy s m|
        s <- ''.
        m <- false.
        taskObj notNil ifTrue:
            [  (bdy <- taskObj body) notNil ifTrue:
                [ s <- s, (bdy printString).
                  m <- true.
            ]   ].
        sleeping ifTrue: [
                s <- s, 's'.
                m <- true.
        ].
        m ifTrue: [ s <- '(', s, ')' ].
        ^name, s.

| printTime     |s mode|
        self setTime.

        s <- name, ' ', (todayWorkTime printHM).
        mode <- ' '.
```

```
        lastFun = 'Working'     ifTrue: [ mode <- 'w' ].
        sleeping                ifTrue: [ mode <- 's' ].
        ^(s, mode)

"----------------       control/query the sleep state."
| setSleeping
        sleeping <- true.

| sleeping
        ^sleeping.

" ---------------- How to start up a worker."
| task: wl create: objClass     | tObj |

        tObj <- self create: objClass.
        tObj startUpTask.
        tObj workers: wl.
        wl do:[ :w |
                w taskObj: tObj.
                w setWorking.
        ].
        ^tObj

| taskObj: t
        taskObj <- t.
| taskObj
        ^taskObj.

| setTime        |t dt|
        t  <- self timeIs.
        dt <- t - lastTime.
        lastFun = 'Idle'
                ifTrue:[
                        idleTime <- idleTime + dt.
                ]ifFalse:[
                        lastFun = 'Working'
                ifTrue: [
                        todayWorkTime <- todayWorkTime + dt.
                        workTime <- workTime + dt.
        ]       ].
        lastTime <- t.

| setTime: fun

        self setTime.
        lastFun <- fun.


| setIdle
        taskObj <- nil.
        self setTime: 'Idle'.

| setWorking
```

```
        self setTime: 'Working'.

| todayWorkTime
        ^todayWorkTime

| hoursWorked   |h m t|
        h <- todayWorkTime hour.
        m <- todayWorkTime min / 60.
        t <- h + m.
        ^t

"---------------- After the rest cycle."
| next
        self pr:[ ((self printString),
                        'is ready for work.')].
        todayWorkTime <- Clock new.
        sleeping <- false.
        self idleWorker: self
]
```

Class GenericTasks

```
     "        Class AnyTask :Task
#        [ startUpTask
#        | max            number of workers allowed.
#        | min            number of workers required
#                                 for one body.
#        | taskTime       return the time required.
#        | prefix: echelon
#        | next
"

"---------------- If There is a truck unload it."
Class Unloader :Task

[ startUpTask
        ^self.
| max             "number of workers allowed."
        ^7.
| min    "number of workers required for one body."
        ^2.
| taskTime       "return the time required."
        ^2.
| prefix: echelon
        ^self name: ( echelon, 'Unload' )
| next
        self setWorking.
        self body: (self unloadTruck: self).
]


"----------------If There is a truck load it."
Class Loader :Task
[ startUpTask
        ^self.
| prefix: echelon
        ^self name: ( echelon, 'Load' )
| max                    "number of workers allowed."
        ^6.
| min            "number of workers required for one body."
        ^2.
| taskTime               "return the time required."
        ^2.
| next
        self setWorking.
        self body:   (self loadTruck: self).
]


"---------------- Process to assign evacuation numbers. "
Class Evac :Task
```

```
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^1.
| min             "number of workers required for one body."
        ^1.
| taskTime                  "return the time required."
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Evac' )
| next
        self setWorking.
        self body:   (self procEvac: self ).
        self reSchedule: self.
]



"------ See who this is and process his personal effects. "
Class Id :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^1.
| taskTime                  "return the time required"
        ^40.
| taskClock | c |
        c _ super taskClock.
        (self body) fingerPrint ifTrue: [ c incMin: 15 ].
        ^c.
| prefix: echelon
        ^self name: ( echelon, 'Id' )
| next
        self setWorking.
        self body:   (self procId: self).
        self reSchedule: self.
]


"-- Pack into transport bag with PE and move to load area."
Class Pack :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^2.
| taskTime                  "return the time required."
        ^10.
| prefix: echelon
```

```
        ^self name: ( echelon, 'Pack' )
| next
        self setWorking.
        self body:   (self procPack: self).
        self reSchedule: self.
]



"---------------- Add him to the convoy list."
Class Dd :Task
[ startUpTask
        ^self.
| max           "number of workers allowed."
        ^1.
| min   "number of workers required for one body."
        ^1.
| taskTime      "return the time required."
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Dd' )
| next
        self setWorking.
        self body: (self procDd: self).
        self reSchedule: self.
]


"----------- Process other overhead items required."
Class Misc :Task
[ startUpTask
        ^self.
| max           "number of workers allowed."
        ^999.
| min   "number of workers required for one body."
        ^1.
| taskTime      "return the time required."
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Misc' )
| next
        self setWorking.
        self body: (self procMisc: self).
        self reSchedule: self.
]


"------ See who this is and process his personal effects. "
Class CkId :Task
[ startUpTask
        ^self.
| max                   "number of workers allowed."
```

```
        ^999.
| min             "number of workers required for one body."
        ^1.
| taskTime                "return the time required"
        ^30.
| prefix: echelon
        ^self name: ( echelon, 'CkId' )
| next
        self setWorking.
        self body:   (self procCkId: self).
        self reSchedule: self.
]


"---------------- Move to load area."
Class Move :Task
[ startUpTask
        ^self.
| max                     "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^2.
| taskTime                "return the time required."
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Move' )
| next
        self setWorking.
        self body:   (self procMove: self).
        self reSchedule: self.
]


"---------------- Remove clothing."
Class   DisRobe :Task
[ startUpTask
        ^self.
| max                     "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^2.
| taskTime                "return the time required"
        ^20.
| prefix: echelon
        ^self name: ( echelon, 'DisRobe' )
| next
        self setWorking.
        self body:   (self procDisRobe: self).
        self reSchedule: self.
]


"---------------- Take finger prints."
```

```
Class   Fingerp :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^1.
| taskTime                  "return the time required"
        ^15.
| prefix: echelon
        ^self name: ( echelon, 'FingerP' )
| next
        self setWorking.
        self body:   (self procFingerP: self).
        self reSchedule: self.
]


"---------------- Do a detailed Id."
Class   DTailId :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^1.
| taskTime                  "return the time required"
        ^30.
| prefix: echelon
        ^self name: ( echelon, 'DTailId' )
| next
        self setWorking.
        self body:   (self procDTailId: self).
        self reSchedule: self.
]


"----------------- Place body into shroud."
Class   Shroud  :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^2.
| taskTime                  "return the time required"
        ^5.
| prefix: echelon
        ^self name: ( echelon, 'Shroud' )
| next
        self setWorking.
        self body:   (self procShroud: self).
        self reSchedule: self.
```

```
]


"---------------- Make Id Plates."
Class   Plates  :Task
[ startUpTask
        ^self.
| max                   "number of workers allowed."
        ^2.
| min           "number of workers required for one body."
        ^1.
| taskTime              "return the time required"
        ^10.
| prefix: echelon
        ^self name: ( echelon, 'Plates' )
| next
        self setWorking.
        self body:   (self procPlates: self).
        self reSchedule: self.
]


"---------------- Dig the Hole, with hole digger."
Class   Dig     :Task
[ startUpTask
        ^self.
| max                   "number of workers allowed."
        ^1.
| min           "number of workers required for one body."
        ^1.
| taskTime              "return the time required"
        ^10.
| prefix: echelon
        ^self name: ( echelon, 'Dig' )
| next
        self setWorking.
        self body:   (self procDig: self).
        self reSchedule: self.
]


"---------------- Prepare personal effects for shipping."
Class   ShipPe  :Task
[ startUpTask
        ^self.
| max                   "number of workers allowed."
        ^999.
| min           "number of workers required for one body."
        ^1.
| taskTime              "return the time required"
        ^15.
| prefix: echelon
        ^self name: ( echelon, 'ShipPe' )
```

```
| next
        self setWorking.
        self body:   (self procShipPe: self).
        self reSchedule: self.
]


"---------------- Place body in grave and cover."
Class   Mv2site :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^2.
| taskTime                  "return the time required"
        ^10.
| prefix: echelon
        ^self name: ( echelon, 'Mv2site' )
| next
        self setWorking.
        self body:   (self procMv2site: self).
        self reSchedule: self.
]

"---------------- Prepare internment and plot records."
Class   Dd3x5   :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^1.
| min             "number of workers required for one body."
        ^1.
| taskTime                  "return the time required"
        ^20.
| prefix: echelon
        ^self name: ( echelon, 'DD3x5' )
| next
        self setWorking.
        self body:   (self procDd3x5: self).
        self reSchedule: self.
]

"---------------- Place body in grave and cover."
Class   InCover :Task
[ startUpTask
        ^self.
| max                       "number of workers allowed."
        ^999.
| min             "number of workers required for one body."
        ^2.
| taskTime                  "return the time required"
        ^15.
```

```
| prefix: echelon
        ^self name: ( echelon, 'InCover' )
| next
        self setWorking.
        self body:   (self procInCover: self).
        self reSchedule: self.
]
```

Copies	Organization
12	Administrator Defense Technical Info Center ATTN: DTIC-DDA Cameron Station Alexandria, VA 22304-6145
1	HQDA DAMA-ART-M Washington, D.C. 20310
1	Commander U.S. Army Materiel Command ATTN: AMCDRA-ST 5001 Eisenhower Avenue Alexandria, VA 22333-0001
1	Commander Armament R&D Center U.S. Army AMCCOM ATTN: SMCAR-TSS Dover, NJ 07801
1	Commander Armament R&D Center U.S. Army AMCCOM ATTN: SMCAR-TDC Dover, NJ 07801
1	Director Benet Weapons Laboratory Armament R&D Center U.S. Army AMCCOM ATTN: SMCAR-LCB-TL Watervliet, NY 12189
1	Commander U.S. Army Armament, Munitions and Chemical Command ATTN: SMCAR-ESP-L Rock Island, IL 61299
1	Commander U.S. Army Aviation Research and Development Command ATTN: AMSAV-E 4300 Goodfellow Blvd St. Louis, MO 63120
1	Director U.S. Army Air Mobility Research and Development Laboratory Ames Research Center Moffett Field, CA 94035

Copies	Organization
1	Commander U.S. Army Communications-Electronics Command ATTN: AMSEL-ED Fort Monmouth, NJ 07703
1	Commander ERADCOM Technical Library ATTN: DELSD-L (Reports Section) Fort Monmouth, NJ 07703-5301
1	Commander U.S. Army Missile Command Research, Development & Engineering Center ATTN: AMSMI-RD Redstone Arsenal, AL 35898
1	Director U.S. Army Missile & Space Intelligence Center ATTN: AIAMS-YDL Redstone Arsenal, AL 35898-550[illegible]
1	Commander U.S. Army Tank Automotive Cmd ATTN: AMSTA-TSL Warren, MI 48397-5000
1	Director U.S. Army TRADOC Systems Analysis Activity ATTN: ATAA-SL White Sands Missile Range, NM 8800[illegible]
1	Commandant U.S. Army Infantry School ATTN: ATSH-CD-CSO-OR Fort Benning, GA 31905
1[illegible]	Commander U.S. Army Development and Employment Agency ATTN: MODE-TED-SAB Fort Lewis, WA 98433
1	AFWL/SUL Kirtland AFB, NM 87117
1	Air Force Armament Laboratory ATTN: AFATL/DLODL Eglin AFB, FL 32542-5000

Copies	Organization
10	Central Intelligence Agency Office of Central Reference Dissemination Branch Room GE-47 HQS Washington, D.C. 20502

ABERDEEN PROVING GROUND

Dir, USAMSAA
ATTN: AMXSY-D
AMXSY-MP, H. Cohen

Cdr, USATECOM
ATTN: AMSTE-TO-F

Cdr, CRDC, AMCCOM
ATTN: SMCCR-RSP-A
SMCCR-MU
SMCCR-SPS-IL

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. BRL Report Number _____ Date of Report _____

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

_____

_____

4. How specifically, is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

_____

_____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided or efficiencies achieved, etc? If so, please elaborate. _____

_____

_____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.)

_____

_____

_____

CURRENT ADDRESS

_____
Name

_____
Organization

_____
Address

_____
City, State, Zip

7. If indicating a Change of Address or Address Correction, please provide the New or Correct Address in Block 6 above and the Old or Incorrect address below.

OLD ADDRESS

_____
Name

_____
Organization

_____
Address

_____
City, State, Zip

(Remove this sheet along the perforation, fold as indicated, staple or tape closed, and mail.)

FOLD HERE

Director
U.S. Army Ballistic Research Laboratory
ATTN: SLCBR-DD-T
Aberdeen Proving Ground, MD 21005-5066

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 12062 WASHINGTON, DC
POSTAGE WILL BE PAID BY DEPARTMENT OF THE ARMY

Director
U.S. Army Ballistic Research Laboratory
ATTN: SLCBR-DD-T
Aberdeen Proving Ground, MD 21005-9989

FOLD HERE

END

2-87

DTIC